Computer Science Department

TECHNICAL REPORT

Synthesizing Linear-Array Algorithms from Nested for Loop Algorithms

P. Lee
Z. Kedem

Technical Report 355

March 1988

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012

Synthesizing Linear-Array Algorithms from Nested for Loop Algorithms

P. Lee
Z. Kedem

Technical Report 355

March 1988

This work was partially supported by the Office of Naval Research under the contract N00014-85-K-0046.

Synthesizing Linear-Array Algorithms from Nested For Loop Algorithms*

PeiZong Lee and Zvi M. Kedem †

Computer Science Department
New York University
251 Mercer Street
New York, NY 10012

March 29, 1988

Abstract

Abstract— This paper is concerned with the mapping of algorithms structured as depth p nested for loops into special purpose systolic VLSI linear arrays. The mappings are done by using linear functions transforming the original sequential algorithms into a form suitable for parallel execution on linear arrays. The derivation of feasible mapping is done by identifying formal criteria to be satisfied by both the original sequential algorithm and proposed transformation function. Those formal criteria define the universe of feasible solutions and thus enable us to derive large families of transformations; the target transformation can be then chosen using additional criteria. Among such criteria could be: minimal execution time, smallest number of processors to be used, or the requirement to use a processor with specific characteristics provided to us. We also study issues dealing with modular extensibility (using one type of processor for arrays of various length) and partitioning (using arrays that are small to solve large problems). The methodology, which deals with general algorithms, is illustrated by synthesizing families of algorithms for matrix multiplication and a version of the Warshall-Floyd transitive closure algorithm.

Index-Terms— VLSI, linear systolic array, algorithm transformations, hyperplane, parallel processing, data dependence, data contention, modularly extensible, partition model, matrix multiplication, path-finding problems.

*This work was partially supported by the Office of Naval Research under the contract N00014-85-K-0046.
†Authors' electronic addresses: leepe@csd2.nyu.edu and kedem@nyu.nyu.edu

1 Introduction

This paper is concerned with designing special purpose VLSI chips to implement particular algorithms. Kung, Hwang and Briggs, Mead and Conway, Ullman and other researchers pointed out that it will be beneficial to design systolic algorithms [16] [24] [10] [36], which are especially suitable for implementation as VLSI chips. For systolic algorithms, consult [16] [9] [33] [18] [19] [21] [42] [6]. Implementation should be as efficient as possible in terms of the space and time resources required. Area-Time trade-offs were studied by Thompson, Brent and Kung, Kedem [35] [5] [11] and others.

A systolic array is a special-purpose parallel device, made out of a few simple processing-element (PE) types whose interconnection pattern exhibits regularity and locality. Several array structures have been proposed, including linear arrays, mesh-connected arrays, and hexagonal arrays. In a typical application, such arrays would be attached as peripheral devices to a host computer which inserts input values into them and extracts output values from them [16].

We will study linear systolic arrays in this paper, as linear arrays are attractive for their bounded I/O requirements and a simple global clock whose rate is independent of the size of the array. We will consider the important class of algorithms structured as (depth) p *nested for loops*. VLSI implementation of such algorithms were studied before, but it is our goal to present a systematic method for transforming them into linear arrays.

The class of a general p nested for loop algorithms includes algorithms to solve matrix multiplication, L-U decomposition, matrix inversion, linear systems, path-finding problem [1] (including all shortest path problem and transitive closure problem), pattern matching, bubble sort, a version of Discrete Fourier Transform (DFT) [24], convolution [16], certain problems solvable by dynamic programming, and others. These problems are of great importance in scientific computation.

Kung [14] [15] [16] first found that the regularity and locality properties of many algorithms make them suitable VLSI systolic array implementations. Among them were matrix multiplication, matrix-vector multiplication, L-U decomposition, and others. Hwang and Cheng, Rote, Ma *et al.*, and Hochet [9] [34] [23] [8] and others found systolic array algorithms for solving linear systems, path-finding problems, matrix inversion, and others. These pioneering works, which increased the understanding of regularity and locality properties of algorithms, were based on the characteristics of an individual algorithm under consideration.

Systematic synthesis methodologies were also proposed. Kung [18], Quinton [30], and Lin and Wah [22] synthesized systolic array directly from uniform recurrent equations. They mapped p nested loop algorithms onto $(p-1)$-D systolic arrays. However, their method was limited in scope because they could not find an easy way to determine the entrance time of input variables and to obtain families of systolic implementations.

Kuhn [12], Moldovan [26] [27] [28] [29], Miranker and Winkler [25], Wong and Delosme [41]

synthesized systolic array based on the hyperplane method introduced by Lamport [20]. They found the data-dependence vectors [20] of an algorithm first, and then they found a nonsingular linear mapping preserving the data-dependence ordering of the original algorithm. This method is also called space-time mapping.

In their model, Kuhn [12] and Moldovan [26] [27] mapped a p dimensional problem space to a t loops 1-D time hyperplane and an s-D space hyperplane mapping, where $p = t + s$. Moldovan [28] [29], and Miranker and Winkler [25] mapped a p dimensional problem space to a 1-D time hyperplane and $(p-1)$-D space hyperplane mapping. Wong and Delosme [41] mapped a p dimensional problem space to a $p-2$ loops 1-D time hyperplane and 2-D space hyperplane mapping, they then mapped this $p-2$ loops 1-D time hyperplane to a 1-D clock ticks.

As shown by examples in those papers, the results were in practice useful for mapping p nested loop algorithms, for $p > 2$, on higher than 1-D arrays. In fact, there is no example given of mapping a p nested loop algorithm into a linear array when $p > 2$.

Another open area left by the previous work was the formulation of necessary and sufficient conditions for correct mappings. As an example we can observe that even though using the previous results it was possible to guarantee that indices are mapped properly on PEs, there was no formal way of guaranteeing that tokens do not clash in the PEs.

Ramakrishnan *et al.* [33] mapped 2-D and 3-D homogeneous graphs on linear arrays. They provided certain necessary conditions for a correct mapping, but they did not provide a complete set of sufficient conditions. As a result, they could not synthesize the linear array implementations of Ramakrishnan and Varman [32].

We now proceed to examine the I/O complexity of common systolic array implementations of the important algorithms we mentioned above (matrix multiplication, etc.). The algorithms were usually mapped into $(p-1)$-D arrays, see works of H. T. Kung, Leiserson, Mead, Conway, Kuhn, Moldovan, Fortes, Rote, Hochet, and S. Y. Kung: [14] [15] [16] [24] [12] [26] [27] [28] [29] [34] [8] [19]. The arrays were of size $O(n^{p-1})$ and the execution time was generally $O(n)$, resulting in optimal processor/time product of $O(n^p)$. However, the number of pins required was at least $\Omega(n^{p-2})$, that is, it increased with the size of the problem solved, if $p > 2$. This may cause potential difficulties during integration of the systolic array with the host computer.

Our goal is to map p nested for loop algorithms on systolic arrays with constant number of pins (bounded I/O requirements), independent of the problem (or array) size. We will thus synthesize linear array systolic algorithms with the following complexity properties. For the set of problems listed above, the execution time will be $O(n^{p-1})$ and the number of I/O pins will be constant. We will need $O(n^{p-1})$ storage locations. There are various way of deciding on the appropriate number of PE's. It is possible to use only $O(n)$ PE's, thus obtaining an optimal processor/time product of $O(n^p)$. This, however, requires storage of $O(n^{p-2})$ in each PE. Thus, if we want the array to

be *modularly extensible*, that is for each PE to have constant amount of storage, we may actually prefer in practice to use $O(n^{p-1})$ PE's.

To accomplish the above in a systematic manner, we present a methodology for mapping p nested for loop algorithms into linear arrays. A mapping is derived by using a function transforming the original sequential algorithm into a form suitable for parallel execution on a linear array. Our approach, similarly to Kuhn's and Moldovan's, is based on Lamport's hyperplane method [20]. However, we transform a p nested for loop algorithm into a 1-D time hyperplane and a 1-D space hyperplane linear-array algorithm.

We also find the data-dependence vectors first. However, we classify them into three types based on certain formal properties. This classification of data-dependence vectors allows us to formulate conditions on the target linear array down to the register level. As data-dependence constraints provide more information than the standard directed graph representations [18] [19] [33], our method can provide more implementations details.

A mapping of a p nested for loop algorithm on a linear array must satisfy certain constraints in order to assure correct flows of token streams. In this paper we list formal *necessary and sufficient conditions* to be satisfied by the mapping assigning tokens to the PE's at various time instances, so that the resulting computation is physically (geometrically) feasible. Those necessary and sufficient conditions are not reducible to the results of Kuhn, Moldovan, Miranker and Winkler, and Wong and Delosme.

This follows from the fact that our necessary and sufficient conditions encompass several different aspects of the design. In addition to preservation of data-dependence and nonsingularity of the mapping (as done by previous researchers) they also include prevention of data contention and complexity of PE's hardware.

As an added benefit, this set of necessary and sufficient conditions allows us to study large classes of mappings, as we are able to find families of solutions satisfying the conditions we derive. Among the feasible solutions we can choose some based on optimality criteria, such as minimum execution time or smallest number of PE's used.

We also analyze time complexity and the storage complexity of linear array implementations. For a class of algorithms we study tight bounds of time complexity of the linear array implementations. Such algorithms include matrix multiplication, L-U decomposition, inversion of nonsingular triangular matrix, matrix orthogonal triangularization, a version of transitive closure algorithm [7], DFT, bubble sort, and others. We also provide a technique to determine whether linear array implementations of these algorithms have both the optimal time complexity and storage complexity.

There may be additional concerns that one may want to consider in choosing a specific linear array. For instance, we may consider relations between the number of PE's and the number of registers in a PE. If the number of registers in a PE is constant, then arrays of various sizes can

be build with a single type of PE, resulting in an *modularly extensible* array. We may also be provided with some "standard" PE type and be required to design a linear array using the specific number of storage cells in the PE's. Our method allows us to understand the relations between such features of the provided PE's and the linear arrays that can be designed.

Finally, we are interested in designing algorithms that can be *partitioned*. Generally, the number of PE's needed for the linear array solving some problem grows with the size of the problem. We may, however, be restricted to arrays of certain size while still being required to solve large problems. To accomplish that, it is necessary to partition the algorithm so that the token streams travel through the array more than once. Partitioning of algorithms was considered before by Guibas *et al.* [7], Hwang and Cheng [9], Moldovan and Fortes [28], Annaratone *et al.* [2], and others. Our method for partitioning is quite general, but we need to impose additional restrictions on the original "unpartitioned" families of solutions before we can partition them. Note that as the unpartitioned algorithm used only a (small) constant numbers of pins, I/O considerations by themselves do not require partitioning as done by e.g. [7] [9] [28].

As implicitly discussed above, it is useful to design linear array implementations for 2 nested loop algorithms [24] [16] [28] [29]. Linear-array algorithms to implement 3 nested for loop algorithms also have been studied before. Some previous results on linear arrays for 3 nested for loop algorithms are due to [13] [17] [31] [32] [33] [40]. The fundamental problem studied there was matrix multiplication. Their results fall within the framework of the methodology described in this paper. In addition, we can also derive families of implementations that were not obtained before, by examining families of solutions satisfying the constraints we define. However, some implementations using different type of architecture, such as busses [38] [39] do not fall within our framework.

The rest of this paper is organized as follows. In Section 2 we introduce the general p nested loop algorithm and the linear-array model we will use. In Section 3 we demonstrate our methodology by synthesizing a matrix-multiplication algorithm. In Section 4, we first introduce the classification of data-dependence vectors which allow us to formulate conditions on the target linear array. Second, we provide the necessary and sufficient conditions so that the mapping results in a correct systolic linear array computation. Third, we analyze time and storage complexity of linear array implementations. For an important class of algorithms, we find tight bounds on the time complexity of linear array implementations. We also provide a technique to show whether the linear array implementations of these algorithms have both the optimal time complexity and optimal storage complexity. In Section 5, we discuss various optimization criteria for general families of algorithms. In Section 6 we discuss the constraints required for partitioning of algorithms. In Section 7 we compare the suitability of two path-finding algorithms for implementation as linear arrays using our methodology. The original Warshall-Floyd path-finding algorithm is not suitable for transformation

to a linear array; however, another reindexed path-finding algorithm due to Kung [19] can be transformed into linear array. Finally, some concluding remarks are given in Section 8. In the Appendix, the methodology of this paper is distilled into a procedure for mapping p nested for loop algorithms into linear arrays.

2 Model for Algorithms and Linear Arrays

2.1 Algorithm Model

In this paper, we consider the class of algorithms with p nested loops of the form:

```
for i_1 = l_1 to u_1 by k_1
    for i_2 = l_2 to u_2 by k_2
            :
        for i_p = l_p to u_p by k_p
            Statement_1 ;
            Statement_2 ;
                :
                :
            Statement_m
        end_for
            :
    end_for
end_for
```

l_j and u_j are integer-valued linear expressions possibly involving $i_1, i_2, \ldots, i_{j-1}$ for $1 < j \leq p$. Without loss of generality, we assume that $l_j \leq u_j$ and $k_j = 1$ for all $1 \leq j \leq p$. When discussing complexity results, it will be more convenient to use one parameter only to describe the size of the problem. Thus we may write $u_i - l_i = O(n)$ for appropriate n.

We make additional assumptions about the structure of the statements inside the loop, as done by Lamport [20], who first studied parallel executions of algorithms written as nested loops. The statements in the loop can be assignment statements, if statements, case statements, or even repetitive statements (including repeat statements, while statements, and for-loop statements). Since the goal of these statements is to compute (modify) some data, each statement can be taken as a special type of assignment statement. These statements contain no I/O statements, no subroutines or functions which can modify data, and no transfer of control to any statement

outside the loop. In addition, innerloop data dependencies have been removed by using compiler techniques as described, for instance, in [4].

Observe that, if we define the index set of the algorithm I^p as

$$I^p = \{(i_1, i_2, \ldots, i_p)^t | l_j \leq i_j \leq u_j \; for \; j = 1, 2, \ldots, p\},$$

then the elements of I^p are ordered in a lexicographical ordering when the loops are executed.

We now formally define our algorithm model, described intuitively earlier.

Definition: A p nested loop algorithm is a 3-tuple $Ag = (I^p, V_{Ag}, F_{Ag})$. Specifically:

1. $I^p = \{(i_1, \ldots, i_p)^t | l_j \leq i_j \leq u_j \; for \; j = 1, \ldots, p\}$ is the loop index set.
2. V_{Ag} is the set of variables.
3. F_{Ag} is the sequence of statements computed in every index in its active phase.

The loop index set and statements satisfy the assumptions discussed above. In addition, when dealing with a p nested loop algorithm, we will frequently write "variable" for the name of an array, or an entry of that array; however, when dealing with a corresponding linear array, we will write "variable" for the name of an array, but we will write "token" for the entry of that array, if this does not result in confusion. For example, we may write variable A, variable $A[17]$, and token $A[17]$.

2.2 Linear Array Model

In this subsection, we present formally the linear-array model we will use. It is a modification of the one employed by Ramakrishnan *et al.* [33]. It may be helpful for the reader to briefly look at Fig. 1 and Fig. 2. As seen there, identical PEs are connected to each other by means of links to form a linear array.

Definition: A linear array is a 5-tuple $Ar = (M, K, T_{Ar}, B_{Ar}, F_{Ar})$, where M is the number of PEs in the array and K, T_{Ar}, B_{Ar} and F_{Ar} form the description of an individual PE. The number K states the number of data links for each PE. Specifically:

1. M is the number of PEs in the array. They will be numbered from left to right as $PE_1, PE_2, ..., PE_M$.
2. K is the number of data links. Every PE has K pairs of input/output ports and such pairs will be referred by integers from $\{1, 2, \ldots, K\}$.

3. $T_{Ar} = (t_1, t_2, \ldots, t_K)$, where $t_i = 1$ or -1 for all $1 \leq i \leq K$, is the sequence of the directions of the (data streams flowing through the) links. If the data stream i flows from left to right then $t_i = 1$, if it flows from right to left then $t_i = -1$.

4. $B_{Ar} = (b_1, b_2, \ldots, b_K)$, where $b_i \geq 0$ for all $1 \leq i \leq K$, is the sequence of the sizes of the buffers (the number of shifting registers) corresponding to the links.

5. F_{Ar} is the K-ary functions computed by every PE in its active phase.

Thus, a linear array has the following communication features:

- Each link i is used for communication between adjacent PEs. Consider link i between PE_j and PE_{j+1} for $1 \leq j < M$, if $t_i = 1$ then the output link i of PE_j is connected to the input link i of PE_{j+1}; if $t_i = -1$ then the output link i of PE_{j+1} is connected to the input link i of PE_j.

- External communication with the host takes place through certain ports designated as follows: for each i if $t_i = 1$ then the input stream i enters I_i of PE_1 and the output is from O_i of PE_M; if $t_i = -1$ then the input stream i enters I_i of PE_M and the output stream is from O_i of PE_1. (I_i's and O_i's are just different terms for links at the boundaries of the array.)

- At time t, if a data token of stream i reaches PE_j, then at time $t + b_i + 1$ this token will reach PE_{j+t_i}, for $1 \leq j + t_i \leq M$.

Each data stream consists of data tokens that travel at a fixed velocity (number of PEs per clock tick) on a unique data link. To implement fixed stream velocity, each data link passes through all the PEs and utilizes a constant number (possibly zero) of shifting registers in each PE as a delay buffer. The buffer lengths are the same in all data links for a specific data stream, so that for a data stream the link delay is the same for all tokens in all PE; however, different data links may have different buffer lengths. A PE is illustrated in Fig. 1 and a linear array is illustrated in Fig. 2.

In Fig. 1, there are $K = l + r$ data streams $1, 2, \ldots, l$ and $l+1, l+2, \ldots, l+r$. The first l data streams flow from left to right and the last r data streams flow from right to left. In Fig. 2, I_i/O_i for $i = 1, \ldots, l$ and I_{l+j}/O_{l+j} for $j = 1, \ldots, r$ are external input/output ports for the $l + r$ data streams $1, 2, \ldots l$ and $l + 1, l + 2, \ldots l + r$ respectiy. Data tokens are fed into and extracted from the array through these ports.

In our model we did not allow $t_i = 0$, though this is of course a simple extension. We are interested here only in implementations in which every data item is pipelined in order to avoid the

need for memory addressing and control hardware in each PE and the preloading and downloading of data. By allowing $t_i = 0$ we can model other implementations, such as WARP [17], in which certain data items are fixed in PEs.

We will relate the F_{Ar} to the F_{Ag} of the algorithm model. In effect, $F_{Ar} = F_{Ag}$.

For each problem "type" such as matrix multiplication we are interested in a family of linear arrays, such that each of this family of linear arrays can solve the problem. Ideally we would like to be also to use a single linear array for all values of the problem size, so that M, K, T_{Ar}, B_{Ar} and F_{Ar} could be made independent of the problem size. However, sometimes we will allow M and b_i's to be functions of the problem size while still insisting that K, T_{Ar} and F_{Ar} are independent of the problem size.

In the next section, we shall show how to map a 3 nested loop algorithm onto a linear array.

3 Example : Matrix Multiplication

We find it expedient to describe the method both formally and by referring to a running example of matrix multiplication. In the Appendix we will state the method concisely.

Let us consider the standard 4×4 matrix-multiplication algorithm.

Input : Matrices $A_{4\times4}$ and $B_{4\times4}$.
Output: Matrix $C_{4\times4}$, where $C_{4\times4} = A_{4\times4} \times B_{4\times4}$.

```
for i = 0 to 3
    for j = 0 to 3
        for k = 0 to 3
            C[i,j] := C[i,j] + A[i,k] * B[k,j]
        end_for
    end_for
end_for
```

Here the algorithm model $Ag = (\{(i,j,k)^t | 0 \leq i,j,k, \leq 3\}, \{A,B,C\}, (C[i,j] := C[i,j] + A[i,k] * B[k,j]))$. Our method consists of several steps.

3.1 The Lamport condition

We reiterate here the fundamental result due to Lamport [20] dealing with parallel execution of nested loops. It was also used by Banerjee *et al.* [4], Kuhn [12], and Moldovan [28]. First, we label each variable in the loop with an index $(i_1, i_2, \ldots, i_p)^t$. We will frequently write "variable" for an

entry of a matrix, or the name of the matrix, if this does not result in confusion. If the variable is on the left-hand-side of :=, it means that the variable is regenerated in index $(i_1, i_2, \ldots, i_p)^t$; if the variable is on the right-hand-side of :=, it means that the variable was previously generated in index $(i_1, i_2, \ldots, i_p)^t$.

For matrix multiplication, $p = 3$. Now, the matrix-multiplication algorithm is described as:

Input : Matrices $A_{4\times4}$ and $B_{4\times4}$.
Output: Matrix $C_{4\times4}$, where $C_{4\times4} = A_{4\times4} \times B_{4\times4}$.

```
1.    for i = 0 to 3
2.        for j = 0 to 3
3.            for k = 0 to 3
4.                A^(i,j,k)[i,k] := A^(i,j-1,k)[i,k];
5.                B^(i,j,k)[k,j] := B^(i-1,j,k)[k,j];
6.                C^(i,j,k)[i,j] := C^(i,j,k-1)[i,j];
7.                C^(i,j,k)[i,j] := C^(i,j,k)[i,j] + A^(i,j,k)[i,k] * B^(i,j,k)[k,j]
8.            end_for
9.        end_for
10.   end_for
```

In line 4, $A^{(i,j,k)}[i,k]$ means that $A[i.k]$ is regenerated in step $(i,j,k)^t$ and $A^{(i,j-1,k)}[i,k]$ means that $A[i,k]$ was previously generated in step $(i,j-1,k)^t$ and is used in step $(i,j,k)^t$. Similarly, for B and C in lines 5 and 6. After having all the needed data, in line 7 it can execute $C[i,j] = C[i,j] + A[i,k] * B[k,j]$.

After labeling, one can define data-dependence vectors. A *data-dependence vector* of a variable can be viewed as difference of indices where a variable is used and where that variable was generated. From lines 4, 5 and 6, it is clear that the index step $(i,j,k)^t$ depends upon all $(i-1,j,k)^t$, $(i,j-1,k)^t$, and $(i,j,k-1)^t$ index steps. Thus, there are three data-dependence vectors in the algorithm:

$d_1 = (0,1,0)^t$ for the pair $(A^{(i,j,k)}[i,k], A^{(i,j-1,k)}[i,k])$
$d_2 = (1,0,0)^t$ for the pair $(B^{(i,j,k)}[k,j], B^{(i-1,j,k)}[k,j])$
$d_3 = (0,0,1)^t$ for the pair $(C^{(i,j,k)}[i,j], C^{(i,j,k-1)}[i,j])$

We say that d_1 is *with* A, d_2 is *with* B, and d_3 is *with* C. In addition, we say that all of d_1, d_2 and d_3 are *related* to C, because $C[i,j]$ uses all $A[i,k]$, $B[k,j]$, and $C[i,j]$, the variables with d_1, d_2,

and d_3, respectively. Therefore, in matrix multiplication we say that index I_2 *depends* on index I_1 if and only if $I_2 = I_1 + m_1 d_1 + m_2 d_2 + m_3 d_3$ for m_1, m_2, $m_3 \geq 0$ and at least one of m_1, m_2, m_3 is positive. These relationships can be described by a data-dependence graph as shown in Fig. 3.

Lamport considered partitioning of indices $(i_1, i_2, \ldots, i_p)^t$ so that they lie on a family of parallel hyperplanes such that all indices lying on one hyperplane can be executed simultaneously. Let $\mathbf{H}$ be a vector $(a_1, a_2, \ldots, a_p)$, then $a_1 x_1 + a_2 x_2 + \cdots + a_p x_p = d$ for various values of d define a family of hyperplanes. We will frequently write "hyperplane" for a family of parallel hyperplanes, the vector defining them, or an individual hyperplane in the family, if this does not result in confusion. A special case of Lamport's result is:

Theorem 1 : *Let* $\mathbf{H} = (a_1, a_2, \ldots, a_p)$ *be a hyperplane. If* $\mathbf{H}d_i > 0$ *for each data-dependence vector* $d_i = (d_{i1}, d_{i2}, \ldots, d_{ip})$ *and two indices* I_1 *and* I_2 *satisfy* $\mathbf{H}I_1 = \mathbf{H}I_2 = c$, *then* I_1 *and* I_2 *are independent of each other, and therefore they can be executed simultaneously.* □

An optimal Lamport's hyperplane $\mathbf{H^L} = (\alpha_1, \alpha_2, \ldots, \alpha_p)$ can be obtained by solving an integer programming problem under the following constraints:

1. $\mathbf{H^L}d_i > 0$ for all data-dependence vectors and
2. $\mathbf{H^L} = \min_{\bar{\mathbf{H}}}\{\max\{|\bar{\mathbf{H}}(I_2 - I_1)| \mid I_1, I_2 \in I^p\}\}$.

$\mathbf{H^L}$ minimizes the number of hyperplanes required to partition the indices and gives an optimal time for parallel implementation of the algorithm on an MIMD (Multiple Instructions and Multiple Data) machine. For matrix multiplication $\mathbf{H^L} = (1, 1, 1)$, that is, the hyperplane is $x + y + z = c$. As we shall immediately see, $\mathbf{H^L}$ defined above cannot be used for implementing linear arrays, as there are locality constraints to be satisfied.

For convenience we will use $\mathbf{H^L}$ to denote Lamport's hyperplanes, $\mathbf{H^{p-1}}$ to denote hyperplanes used for mappings on $(p-1)$-D arrays, and $\mathbf{H^1}$ to denote hyperplanes used for mappings on 1-D (linear) arrays. Subscripts may be used to distinguish between different hyperplanes in the same "class."

3.2 Computation of $\mathbf{H^1}$ and S

It is our goal to assign each index of I^p to both a specific time instance and a specific linear array location by means of a linear transformation. We can therefore describe the desired assignment as a linear mapping from p dimensions into 2 dimensions. Thus $(i_1, i_2, \ldots, i_p)^t \longmapsto (t, l)$ where t specifies the time instance and l the PE number. We refer to this mapping as a 1-D time hyperplane and a 1-D space hyperplane linear-array algorithm $(\mathbf{H^1}, \mathbf{S})$. This terms will be explained in the next paragraph. As finding the mapping is the heart of the method, the description of this step will

be quite long and detailed. We will also state precisely in Section 4 the conditions to be satisfied for correct and efficient implementation.

First, we define $\mathbf{H^1}$ and S. $\mathbf{H^1}$ is a vector $(h_1, h_2, \ldots, h_p)$ and S is a vector $(s_1, s_2, \ldots, s_p)$. Given an index $(i_1, i_2, \ldots, i_p)^t$, the mapping $\begin{pmatrix} \mathbf{H^1} \\ \mathbf{S} \end{pmatrix} (i_1, i_2, \ldots, i_p)^t$ results in a 2 dimensional vector:

$$\begin{pmatrix} h_1 & h_2 & \cdots & h_p \\ s_1 & s_2 & \cdots & s_p \end{pmatrix} (i_1, i_2, \ldots, i_p)^t = \begin{pmatrix} h_1 i_1 + h_2 i_2 + \cdots + h_p i_p \\ s_1 i_1 + s_2 i_2 + \cdots + s_p i_p \end{pmatrix} = \begin{pmatrix} t \\ l \end{pmatrix}$$

where t and l specify the time and the location of a data token with index $(i_1, i_2, \ldots, i_p)^t$.

For completeness, we relate the above to work done on mapping nested loops to (p-1)-D arrays. As mentioned previously this was studied by Kuhn [12] and Moldovan [28]. They mapped the p-D space of indices into a p-D space: $(i_1, i_2, \ldots, i_p)^t \longmapsto (t, l_1, \ldots, l_{p-1})$, where $(l_1, \ldots, l_{p-1})$ state the location of the PE executing index $(i_1, i_2, \ldots, i_p)^t$ at time t.

Their algorithms, which we denote by $(\mathbf{H^{p-1}}, \mathbf{S^{p-1}})$, can therefore be described by means of a mapping described by p nested loops on a p dimensional space. $\mathbf{H^{p-1}}$ is a vector $(h_1^{p-1}, \ldots, h_p^{p-1})$ and $\mathbf{S^{p-1}}$ is a matrix $\begin{pmatrix} s_{11} & \cdots & s_{1p} \\ \vdots & \ddots & \vdots \\ s_{(p-1)1} & \cdots & s_{(p-1)p} \end{pmatrix}$. Given an index $(i_1, i_2, \ldots, i_p)^t$, the mapping $\begin{pmatrix} \mathbf{H^{p-1}} \\ \mathbf{S^{p-1}} \end{pmatrix} (i_1, i_2, \ldots, i_p)^t$ is a p dimensional vector:

$$\begin{pmatrix} h_1^{p-1} & \cdots & h_p^{p-1} \\ s_{11} & \cdots & s_{1p} \\ \vdots & \ddots & \vdots \\ s_{(p-1)1} & \cdots & s_{(p-1)p} \end{pmatrix} (i_1, i_2, \ldots, i_p)^t = \begin{pmatrix} h_1^{p-1} i_1 + \cdots + h_p^{p-1} i_p \\ s_{11} i_1 + \cdots + s_{1p} i_p \\ \vdots \quad + \quad \ddots \quad + \quad \vdots \\ s_{(p-1)1} i_1 + \cdots + s_{(p-1)p} i_p \end{pmatrix} = \begin{pmatrix} t \\ l_1 \\ \vdots \\ l_{p-1} \end{pmatrix}$$

where t and $(l_1, \ldots, l_{p-1})$ specify the time and the (p-1)-D location of a data token with index $(i_1, i_2, \ldots, i_p)^t$. When given a $\mathbf{H^L}$, they found a space mapping $\mathbf{S^{p-1}}$ by requiring that $\begin{pmatrix} \mathbf{H^L} \\ \mathbf{S^{p-1}} \end{pmatrix}$ be nonsingular. When given restricted interconnection primitives and $\mathbf{S^{p-1}}$ first, they sometimes also did not use $\mathbf{H^L}$, but had to use another hyperplane $\mathbf{H^{p-1}}$ that would make $\begin{pmatrix} \mathbf{H^{p-1}} \\ \mathbf{S^{p-1}} \end{pmatrix}$ nonsingular.

As claimed above, our problem will be more difficult, as in general we will <u>not</u> be able to find any S corresponding to a 1-D array with $\mathbf{H^1} = \mathbf{H^L}$.

Note that as all space hyperplanes will deal with 1-D arrays, we will write S instead of possibly more appropriate $\mathbf{S^1}$.

3.3 The example continued

We now continue with our example. As stated above $\mathbf{H^L} = (1,1,1)$, that is, the hyperplane is $x + y + z = c$. Now let us consider a space hyperplane $\mathbf{S} = (s_1, s_2, s_3)$ such that all the indices in a hyperplane $s_1x + s_2y + s_3z = c$ are mapped into PE_c of the linear array. For specificity of explanation, suppose for now that the space hyperplane is $\mathbf{S} = (1,1,-1)$. As shown in Fig. 4, there are ten indices on the time hyperplane ($\mathbf{H^L}$) $x + y + z = 3$: $(0,0,3)^t$, $(0,1,2)^t$, $(0,2,1)^t$, $(0,3,0)^t$, $(1,0,2)^t$, $(1,1,1)^t$, $(1,2,0)^t$, $(2,0,1)^t$, $(2,1,0)^t$, and $(3,0,0)^t$. Let us now consider on which PEs those indices are mapped. For convenience, we will assume that the PEs are numbered from -3 to 6, and not from 1 to M as formally required by our model. Since the space hyperplane is $\mathbf{S} : x + y - z = c$, then

$(0,3,0)^t$,$(1,2,0)^t$,$(2,1,0)^t$, and $(3,0,0)^t$	will be mapped to	PE_3,
$(0,2,1)^t$,$(1,1,1)^t$, and $(2,0,1)^t$	:	PE_1,
$(0,1,2)^t$ and $(1,0,2)^t$	:	PE_{-1},
$(0,0,3)^t$	:	PE_{-3}.

However, we must prevent $(0,3,0)^t$, $(1,2,0)^t$, $(2,1,0)^t$, and $(3,0,0)^t$ from being mapping to PE_3 at the same time instance, here time instance 3. (Similarly, for $(0,2,1)^t$, $(1,1,1)^t$, and $(2,0,1)^t$ to PE_1 as well as for $(0,1,2)^t$ and $(1,0,2)^t$ to PE_{-1}.)

Thus, we need to use several different time hyperplanes, at least for this space hyperplane. The reader will wonder at this point what would happen for space hyperplanes other than our example space hyperplane $(1,1,-1)$. This will be discussed formally later in the paper.

It is our goal to find a time hyperplane $\mathbf{H^1}$ that will allow assignment of at most one index to a PE at any time instance. In order to find such time hyperplane, various methods might be used. We found the most expedient to show how to derive $\mathbf{H^1}$ by modifying Lamport's $\mathbf{H^L}$. Thus, our $\mathbf{H^1}$ will be written as a linear combination: $\mathbf{H^1} = \mathbf{H^L} + \mathbf{\Pi}$, where $\mathbf{\Pi}$ is referred to as an assistant hyperplane. Our notation will be:

$\mathbf{H^L}$	=	$(\alpha_1, \alpha_2, \ldots, \alpha_p)$	is Lamport's hyperplane,
$\mathbf{\Pi}$	=	$(\pi_1, \pi_2, \ldots, \pi_p)$	is our assistant hyperplane,
$\mathbf{H^1}$	=	$\mathbf{H^L} + \mathbf{\Pi}$	is our time hyperplane, and
$\mathbf{S}$	=	$(s_1, s_2, \ldots, s_p)$	is our space hyperplane.

We now continue with the four indices: $(0,3,0)^t$, $(1,2,0)^t$, $(2,1,0)^t$, and $(3,0,0)^t$ which were mapped into PE_3 at time instance 3. To "spread" them in time, we introduce an assistant hyperplane $\mathbf{\Pi} = (\pi_1, \pi_2, \pi_3)$ that cuts the hyperplane $\mathbf{H^L}$ so that each time at most one of these four indices will be mapped to PE_3. Suppose the assistant hyperplane is $\mathbf{\Pi} : x + 2z = d$. Then in the time hyperplane $x + y + z = 3$,

	$(0,3,0)^t$	will be executed at	$x+2z=0$,
	$(1,2,0)^t$	:	$x+2z=1$,
	$(0,2,1)^t$ and $(2,1,0)^t$	:	$x+2z=2$,
	$(1,1,1)^t$ and $(3,0,0)^t$	:	$x+2z=3$,
	$(0,1,2)^t$ and $(2,0,1)^t$	:	$x+2z=4$,
	$(1,0,2)^t$	:	$x+2z=5$,
and	$(0,0,3)^t$	:	$x+2z=6$.

At this point $(0,3,0)^t$, $(1,2,0)^t$, $(2,1,0)^t$, and $(3,0,0)^t$ will not be mapped into PE_3 at the same time. (Similarly, for $(0,2,1)^t$, $(1,1,1)^t$, and $(2,0,1)^t$ to PE_1 as well as for $(0,1,2)^t$ and $(1,0,2)^t$ to PE_{-1}.)

However, if $\mathbf{H^L}$ and $\mathbf{\Pi}$ are linearly independent and $(\mathbf{H^L}+\mathbf{\Pi})d_i > 0$ for all data-dependence vectors, then we can overlap to execute the indices according to $\mathbf{H^L}+\mathbf{\Pi}$ (see Fig. 5). In fact, if $\mathbf{\Pi}$ slices $\mathbf{H^L}$, $\mathbf{H^L}$ and $\mathbf{\Pi}$ must be linearly independent. In addition, if $(\mathbf{H^L}+\mathbf{\Pi})d_i > 0$ for all data-dependence vectors, then $\mathbf{H^L}+\mathbf{\Pi}$ still satisfies Theorem 1. So let the new hyperplane $\mathbf{H^1} = \mathbf{H^L}+\mathbf{\Pi} = (\alpha_1,\alpha_2,\alpha_3)+(\pi_1,\pi_2,\pi_3) = (1,1,1)+(1,0,2) = (2,1,3)$, that is, $\mathbf{H^1} : 2x+y+3z = e$.

Then	$(0,3,0)^t$	will be executed at	$2x+y+3z=3$,
	$(1,2,0)^t$	:	$2x+y+3z=4$.
	$(0,2,1)^t$ and $(2,1,0)^t$	:	$2x+y+3z=5$,
	$(1,1,1)^t$ and $(3,0,0)^t$	:	$2x+y+3z=6$,
	$(0,1,2)^t$ and $(2,0,1)^t$	:	$2x+y+3z=7$,
	$(1,0,2)^t$	:	$2x+y+3z=8$,
and	$(0,0,3)^t$	:	$2x+y+3z=9$.

Recall that from the discussion above, indices $(0,2,1)^t$ and $(2,1,0)^t$ are executed in different PEs. (Similarly, for $(1,1,1)^t$ and $(3,0,0)^t$ as well as for $(0,1,2)^t$ and $(2,0,1)^t$.) All the indices will be mapped to the PEs according to the new time hyperplane $\mathbf{H^1} = (2,1,3)$ and the space hyperplane $\mathbf{S} = (1,1,-1)$ as in Fig. 6.

This mapping maps index $(i,j,k)^t$ into $PE_{\mathbf{S}(i,j,k)^t} = PE_{(i+j-k)}$ and executes it at step $\mathbf{H^1}(i,j,k)^t = (2i+j+3k)$. We now describe the behavior of the resulting linear array:

1. The tokens of A and B are pipelined and enter into the linear array from left to right; the tokens of C are pipelined and enter into the linear array from right to left. The tokens of A are fed into data link 1, the tokens of B are fed into data link 2, and the tokens of C are fed into data link 3. (The entrance times of the tokens of A, B and C will be computed later.)

2. We now consider the speed of the token streams. Formal conditions will be given later. For now we examine Fig. 6. The tokens of A flow at full speed, that is, there is no delay for the tokens of A. The tokens of B flow at half speed, that is, there is one unit time delay when a token enters a PE, or say, there is a delay buffer with one shifting register for the data link

of B. Finally, the tokens of C flow at one third speed, that is, there are two units of time delay, or say, there is a delay buffer with two shifting registers for the data link of C.

3. When the three tokens $A[i,k]$, $B[k,j]$ and $C[i,j]$ enter $PE_{(i+j-k)}$ at time $2i+j+3k$, the PE executes the instruction $C[i,j] := C[i,j] + A[i,k] * B[k,j]$. □

In this example, we found a time hyperplane $\mathbf{H^1}$ and a space hyperplane $\mathbf{S}$ so that they map the matrix multiplication algorithm to the linear array. In general, the time hyperplane $\mathbf{H^1}$ and the space hyperplane $\mathbf{S}$ must satisfy certain constraints which we will state next.

4 On Synthesis of Linear Array Algorithms

In this section, we will state formal necessary and sufficient conditions to be satisfied by the mapping $(\mathbf{H^1}, \mathbf{S})$ for correct implementation of p nested for loop algorithms on linear arrays.

Consider some p nested loop algorithm. After labeling ("labeling" was introduced in Subsection 3.1), the number of statements in the loop body increases. However, all these additional statements are trivial assignment statements (for example, $x := x$;), as they are used only for defining the data-dependence vectors.

Let $D_{Ag} = (d_1, d_2, \ldots, d_w)$ be the sequence of the data-dependence vectors. Each data dependence vector is with a single specific variable in V_{Ag} ("with" was defined in Section 3) and each variable is also with at least one data-dependence vector ("with" is symmetric here). There may be several data-dependence vectors with a variable; however, no two variables are with the same data-dependence vector. (Thus we allow $d_i = d_j$ for $i \neq j$.) We make a non-essential simplifying assumption. We assume that if $d_i = (d_{i1}, d_{i2}, \ldots, d_{ip})$, then $\gcd(d_{i1}, d_{i2}, \ldots, d_{ip}) = 1$. The case when $\gcd(d_{i1}, d_{i2}, \ldots, d_{ip}) \neq 1$ can be handled by a simple extension.

We will now relate the algorithm Ag to an array Ar. In our mapping we will need w data links, as we will associate a dedicated data link with each data-dependence vector. Thus, in general $K \geq w$, and for simplicity we assume that $K = w$. d_i will correspond to some data link i' of the linear-array model. For simplicity, we assume that $i' = i$.

We now describe the relation between the variables of Ag and the w data links. If a variable $V \in V_{Ag}$ is with some number $\delta(V)$ data-dependence vectors, we will dedicate $\delta(V)$ links to it. In effect, $\delta(V)$ copies of that variable will be "traveling" through the array, each in a dedicated data link. A variable, in general, is an array and therefore may consist of many "atomic" entries. In each data link dedicated to the variable, all those entries will appear during the execution of the algorithm. Formally, we will say that several *data streams* (one per data link) are associated with each variable and each data stream consists of all the *tokens* (individual atomic entries) of the variable.

To illustrate this, consider some algorithm in which the array $A[1..5, 1..5]$ is with three data-dependence vectors d_α, d_β, and d_γ. Then there are three data links dedicated to A. Each entry of A, e.g. $A[1,4]$ participates in three data streams. In effect, we have three data streams, of 25 tokens each. Thus, A gives rise to 75 tokens.

We will now discuss how the properties of the algorithm Ag and the corresponding data dependence vectors D_{Ag} influence the structure and the behavior of the linear array Ar implementing Ag. Consider some data dependence vector d_i with some variable X. Let x be any token of the data stream corresponding to d_i. Then based on the behavior of Ag, we can classify d_i as being of one of the three types:

type 1 : $d_i \neq 0$ and the token x is used (and regenerated) in all indices of I^p of the form $\bar{I} + md_i$ for some $\bar{I} \in I^p$ and all integers m.

type 2 : $d_i \neq 0$ and the token x is used (on the right hand side of :=) in $\bar{I} \in I^p$ and is generated once only in $\bar{I} - d_i \in I^p$.

type 3 : $d_i = 0$. Either x is generated (on the left hand side of :=) once only, or x is only used (on the right hand side of :=) but it is not generated (or regenerated) in any index (X is an input variable). (As we will discuss later, this implies that the computed value of x will not be used in data stream i, or the token x is used only once in data stream i but is not generated in any data stream.)

Lemma 2 *(Zero-One-Infinite)* : *The three types are exhaustive, that is, no other case is possible.* □

In order to discuss these three types, we proceed to an example in which they all occur. Consider the following (Longest Common Subsequence) algorithm:

```
Input : Arrays A[1..m] and B[1..n].
Output: Matrix C[1..m,1..n], where C[i,j] = the length of the longest
        common subsequence of A[1..i] and B[1..j].
    for i = 1 to m
        for j = 1 to n
            if A[i] = B[j]
            then C[i,j] := C[i-1,j-1] + 1
            else C[i,j] := max{C[i,j-1], C[i-1,j]}
        end_for
    end_for
```

Here the algorithm model $Ag = (\{(i,j)^t | 1 \leq i \leq m,\ 1 \leq j \leq n\}, \{A, B, C\}$, (**if** $A[i] = B[j]$ **then** $C[i,j] := C[i-1, j-1] + 1$ **else** $C[i,j] := \max\{C[i, j-1], C[i-1, j]\}))$. There are six variables (i.e. $A[i]$, $B[j]$, $C[i-1, j-1]$, $[i, j-1]$, $C[i-1, j]$, and $C[i,j]$) in the loop body. After labeling, the algorithm becomes:

Input : Arrays $A[1..m]$ and $B[1..n]$.
Output: Matrix $C[1..m, 1..n]$, where $C[i,j]$ = the length of the longest
common subsequence of $A[1..i]$ and $B[1..j]$.

for $i = 1$ to m
for $j = 1$ to n
1. $A^{(i,j)}[i] := A^{(i,j-1)}[i]$;
2. $B^{(i,j)}[j] := B^{(i-1,j)}[j]$;
3. $C^{(i,j)}[i-1, j-1] := C^{(i-1,j-1)}[i-1, j-1]$;
4. $C^{(i,j)}[i, j-1] := C^{(i,j-1)}[i, j-1]$;
5. $C^{(i,j)}[i-1, j] := C^{(i-1,j)}[i-1, j]$;
6. $C^{(i,j)}[i,j] := C^{(i,j)}[i,j]$;
7. **if** $A^{(i,j)}[i] = B^{(i,j)}[j]$
then $C^{(i,j)}[i,j] := C^{(i,j)}[i-1, j-1] + 1$
else $C^{(i,j)}[i,j] := \max\{C^{(i,j)}[i, j-1], C^{(i,j)}[i-1, j]\}$
end_for
end_for

From lines 1 to 6. we get six data-dependence vectors,

$d_1 = (0,1)^t$ for $(A^{(i,j)}[i], A^{(i,j-1)}[i])$,
$d_2 = (1,0)^t$ for $(B^{(i,j)}[j], B^{(i-1,j)}[j])$,
$d_3 = (1,1)^t$ for $(C^{(i,j)}[i-1, j-1], C^{(i-1,j-1)}[i-1, j-1])$,
$d_4 = (0,1)^t$ for $(C^{(i,j)}[i, j-1], C^{(i,j-1)}[i, j-1])$,
$d_5 = (1,0)^t$ for $(C^{(i,j)}[i-1, j], C^{(i-1,j)}[i-1, j])$, and
$d_6 = (0,0)^t$ for $(C^{(i,j)}[i,j], C^{(i,j)}[i,j])$.

d_1 and d_2 are of type 1, d_3, d_4, and d_5 are of type 2, and d_6 is of type 3.

If a token is with a type 1 data-dependence vector, then this token is needed throughout the execution. In our example, both $A[i]$ (with d_1) and $B[j]$ (with d_2) are such tokens.

If a token is with a type 2 data-dependence vector, then this token is not needed after it is used once. This observation points out that a token with type 2 data-dependence vector will not be an input token or an output token. For example, $C[i-1, j-1]$, which was generated in index $(i-1, j-1)^t$, is with d_3. Thus, $C[i-1, j-1]$ can be destroyed after it was used in index $(i, j)^t$. Similarly, $C[i, j-1]$, which was generated in index $(i, j-1)^t$, is with d_4 and can be destroyed after it was used in index $(i, j)^t$. Finally, $C[i-1, j]$, which was generated in index $(i-1, j)^t$, is with d_5 and can be destroyed after it was used in index $(i, j)^t$.

As we excluded the case when $Sd_i = 0$ in this paper, we will not consider the case when $d_i = 0$ in the rest of the paper. For completeness sake, however, we will present a very brief discussion. If a token is with a type 3 data-dependence vector, its behavior is more complex. It is generated (in the data stream corresponding to d_i) but never used again in that data stream. Its value, however, is not lost but must be copied as input value for other data streams. For example, $C[i, j]$ is copied to data links 3, 4, and 5 (corresponding to data streams 3, 4, and 5).

If some $d_i = 0$, in the resulting array each PE will need an I/O port for transferring tokens between the host computer. Thus, the total number of I/O ports will not be constant. As stated in the introduction, in this paper we are interested in linear arrays with a constant number of I/O ports.

We now discuss an important property of data streams. Each data stream of variable X with a specific data-dependence vector d_i has only one token (entry of X) used (and regenerated) in each index. Therefore, each assignment statement in the loop body can be seen as a w-ary function. For example, $r = \hat{F}_{Ag}(v_1, v_2, \ldots, v_w)$, where $\hat{F}_{Ag}$ is a statement in F_{Ag} and v_i is an entry of variable V_i, which is with d_i.

Since all the statements in F_{Ag} can be handled in the same way, in the following we only consider a single, representative, statement $\hat{F}_{Ag}$ instead of a sequence of statements F_{Ag}. For simplicity, we assume that only one time unit is needed to execute the whole statements in the loop body.

A correct linear array algorithm $(\mathbf{H^1}, \mathbf{S})$ that maps a p nested loop algorithm Ag into a linear array Ar must preserve data-dependence relations, the right tokens must be in the right place at the right time, and in addition, data tokens must not collide in data links.

It is our goal to find necessary and sufficient conditions on synthesizing such linear array algorithms $(\mathbf{H^1}, \mathbf{S})$. We are also interested in the time complexity and the storage complexity for $(\mathbf{H^1}, \mathbf{S})$. Furthermore, for a special class of algorithms, whose set of the data-dependence vectors is $\{d_1, d_2, \ldots, d_w\} = \{(1, 0, \ldots, 0)^t, (0, 1, \ldots, 0)^t, \ldots, (0, 0, \ldots, 1)^t\}$, we are interested in tight bounds on time complexity of their linear array implementations. This class of algorithms is of particular interest, as it includes our example — matrix multiplication, and algorithms to solve L-U decomposition, inversion of nonsingularly triangular matrix, matrix orthogonal triangularization, a version of transitive closure algorithm [7], DFT, bubble sort [21], and others.

4.1 Necessary Conditions

We now consider conditions to be satisfied by $(\mathbf{H}^1, \mathbf{S})$ in order to assure a correct construction. There are five necessary conditions for $(\mathbf{H}^1, \mathbf{S})$.

1. $\mathbf{H}^1$ has to preserve the data-dependence relation, that is, if $I_2 - I_1 = d_i$ for any two indices I_1 and I_2 then I_2 must be executed after I_1. That is, $\mathbf{H}^1 I_2 - \mathbf{H}^1 I_1 > 0$, or, $\mathbf{H}^1(I_2 - I_1) = \mathbf{H}^1 d_i > 0$.

Condition 1 : $\mathbf{H}^1 d_i > 0$ *must be true for all data-dependence vectors* d_i.

Note: This follows immediately from Lamport's result [20].

2. No two indices I_1 and I_2 can be mapped to the same PE at the same time, that is, $\mathbf{H}^1 I_1 = \mathbf{H}^1 I_2$ and $\mathbf{S} I_1 = \mathbf{S} I_2$ can not be both true at the same time.

Condition 2 : *If* I_1 *and* I_2 *are two indices of* I^p *then*

$$\begin{pmatrix} \mathbf{H}^1 \\ \mathbf{S} \end{pmatrix} (I_2 - I_1) \neq \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \text{ that is, } \begin{pmatrix} \mathbf{H}^1 \\ \mathbf{S} \end{pmatrix} (x_1, \ldots, x_p)^t \neq \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

for $(l_j - u_j) \leq x_j \leq (u_j - l_j)$ *and* $j = 1, \ldots, p$.

Note: This Condition is related to the nonsingularity condition of Kuhn [12] and Moldovan [28]. Varman and Ramakrishnan [40] also gave a special case of Condition 2 to construct an array for matrix multiplication.

3. Next, suppose a variable with the data-dependence vector d_i is generated in index $\bar{I}$ and will be used next time in index $\bar{I} + d_i$. Index $\bar{I}$ is executed in $PE_{\mathbf{S}\bar{I}}$ at time $\mathbf{H}^1 \bar{I}$, and the index $\bar{I} + d_i$ will be executed in $PE_{\mathbf{S}(\bar{I}+d_i)}$ at time $\mathbf{H}^1(\bar{I} + d_i)$. Thus the corresponding token is in $PE_{\mathbf{S}\bar{I}}$ at time $\mathbf{H}^1 \bar{I}$ and is in $PE_{\mathbf{S}(\bar{I}+d_i)}$ at time $\mathbf{H}^1(\bar{I} + d_i)$. Therefore, $\frac{\mathbf{H}^1(\bar{I}+d_i) - \mathbf{H}^1 \bar{I}}{\mathbf{S}(\bar{I}+d_i) - \mathbf{S}\bar{I}} = \frac{\mathbf{H}^1 d_i}{\mathbf{S} d_i}$ must be an integer, as this token is delayed by a constant amount of time in each PE. (Note: if $\mathbf{S} d_i > 0$ then $\frac{\mathbf{H}^1 d_i}{\mathbf{S} d_i}$ is positive, and that token is delayed by $\frac{\mathbf{H}^1 d_i}{\mathbf{S} d_i}$ time. If $\mathbf{S} d_i < 0$ then $\frac{\mathbf{H}^1 d_i}{\mathbf{S} d_i}$ is negative, and that token is delayed by $-\frac{\mathbf{H}^1 d_i}{\mathbf{S} d_i}$ time.)

We now examine the data flow behavior of data stream i. Consider a token with data dependence vector d_i. From the discussion above, it follows that it will be delayed for $|\frac{\mathbf{H}^1 d_i}{\mathbf{S} d_i}|$ time while travelling through one PE. One time unit is allocated to the processing time and therefore we have to account for the remaining $|\frac{\mathbf{H}^1 d_i}{\mathbf{S} d_i}| - 1$ time units. To accomplish that, we need $|\frac{\mathbf{H}^1 d_i}{\mathbf{S} d_i}| - 1$ shifting

registers in the data link i.

Condition 3 : b_i, *the number of shifting registers in each PE for the data link* i, *must be* $|\frac{\mathbf{H^1}d_i}{\mathbf{S}d_i}|-1$.

Sometimes, when we wish to refer directly to a condition on the value of $\frac{\mathbf{H^1}d_i}{\mathbf{S}d_i}$, we may use Condition 3′:

Condition 3′ : $\frac{\mathbf{H^1}d_i}{\mathbf{S}d_i}$ *must be an integer for all data-dependence vectors* d_i.

4. Now, define the positive direction of the linear array to be from left to right and the negative direction to be from right to left. Then, a token with data-dependence vector d_i that is generated in index $\bar{I}$ is executed in $PE_{\mathbf{S}\bar{I}}$ at time $\mathbf{H^1}\bar{I}$ and will be used in index $\bar{I}+d_i$ in $PE_{\mathbf{S}(\bar{I}+d_i)}$ at time $\mathbf{H^1}(\bar{I}+d_i)$. Since $\mathbf{H^1}(\bar{I}+d_i) > \mathbf{H^1}\bar{I}$, if $\mathbf{S}d_i > 0$ then $PE_{\mathbf{S}(\bar{I}+d_i)} > PE_{\mathbf{S}\bar{I}}$ and the token will flow from $PE_{\mathbf{S}\bar{I}}$ to $PE_{\mathbf{S}(\bar{I}+d_i)}$ in the positive direction, and if $\mathbf{S}d_i < 0$ then $PE_{\mathbf{S}(\bar{I}+d_i)} < PE_{\mathbf{S}\bar{I}}$ and the token will flow from $PE_{\mathbf{S}\bar{I}}$ to $PE_{\mathbf{S}(\bar{I}+d_i)}$ in the negative direction. Let $t_i = 1$ denote that the data stream i has the positive direction, and let $t_i = -1$ denote that the data stream i has the negative direction. Then we restate the above as the condition:

Condition 4 : *If* $\mathbf{S}d_i > 0$, *then* $t_i = 1$ *and the data stream* i *will be fed into the linear array at* $PE_{\min\{\mathbf{S}I_1 | I_1 \in I^p\}}$. *If* $\mathbf{S}d_i < 0$, *then* $t_i = -1$ *and the data stream* i *will be fed into the linear array at* $PE_{\max\{\mathbf{S}I_2 | I_2 \in I^p\}}$.

Note: (1) Ramakrishnan *et al.* [33] also gave a similar condition to Condition 4, but they did not base it on the data-dependence vector d_i. (2) If $\mathbf{S}d_i = 0$ then $t_i = 0$ and the data stream will be fixed in the PEs. However, as stated above, we do not consider $\mathbf{S}d_i = 0$ in this paper.

Lemma 3 : *Let* $(\mathbf{H^1}, \mathbf{S})$ *satisfy Condition 3 and Condition 4 and let token* x *be with* d_i. *If* x *is in* PE_a *at time* T_a *and will be in* PE_b *at time* T_b, *then* $\frac{T_b - T_a}{b-a} = t_i(b_i+1) = \frac{\mathbf{H^1}d_i}{\mathbf{S}d_i}$.

Proof : Immediately from Conditions 3 and 4. □

Corollary 4 : *Let* $(\mathbf{H^1}, \mathbf{S})$ *satisfy Condition 3 and Condition 4 and let token* x *be with* d_i. *If* x *is used in index* $\bar{I}$ *in* $PE_{\mathbf{S}\bar{I}}$ *at time* $\mathbf{H^1}\bar{I}$ *and will be used in index* $\bar{I}+d_i$, *then* x *will be in* $PE_{\mathbf{S}(\bar{I}+d_i)}$ *at time* $\mathbf{H^1}(\bar{I}+d_i)$.

Proof : Immediately from Lemma 3. □

We can now derive the entrance time of each data token.

Corollary 5 : *Suppose token x, whose data-dependence vector is d_i of type 1, arrives at some PE_a at time T_a.*
If $\mathbf{S}d_i > 0$, then x is fed into the linear array into $PE_{\min\{\mathbf{S}\bar{I}|\bar{I}\in I^p\}}$ at time

$$T_a - (a - \min\{\mathbf{S}\bar{I}|\bar{I} \in I^p\})\frac{\mathbf{H}^1 d_i}{\mathbf{S}d_i}.$$

If $\mathbf{S}d_i < 0$, then x is fed into $PE_{\max\{\mathbf{S}\bar{I}|\bar{I}\in I^p\}}$ at time

$$T_a - (\max\{\mathbf{S}\bar{I}|\bar{I} \in I^p\} - a)\frac{\mathbf{H}^1 d_i}{\mathbf{S}d_i}.$$

Proof : From Lemma 3, by linear interpolation. □

Corollary 5 shows that once we know that x enters some PE at some time, then we can obtain the entrance time of x. On the other hand, if x enters the linear array at the specific entrance time, then x will be in that PE at that time.

5. It may seem that we have identified all "major" necessary conditions. This, however, is not the case. So far we have only examined the behavior of individual tokens. Now we will consider the possible interference between tokens of the same data stream.

Let x_1 and x_2 be tokens of the variable X in the data stream corresponding to d_i. (Thus, of course, d_i is with X.) x_1 is used in index I_1 and x_2 is used in index I_2. Conceivably $x_1 = x_2$. To characterize when $x_1 \neq x_2$, we have two lemmas, depending on the type of d_i. As the case where it is of type 2 is trivial, we deal with it first.

Lemma 6 : *Let x_1 and x_2 be tokens of variable X in the data stream corresponding to d_i, which is of type 2. Let x_1 be used in some index I_1 and let x_2 be used in some index I_2. Then, $x_1 \neq x_2$ if and only if $I_1 \neq I_2$.*

Proof : Immediately from the fact that each token is used in one index only, and no two tokens of any data stream are used in the same index. □

Lemma 7 : *Let x_1 and x_2 be tokens of variable X in the data stream corresponding to d_i, which is of type 1. Let x_1 be used in some index I_1 and let x_2 be used in some index I_2. Then, $x_1 \neq x_2$ if and only if $I_2 - I_1$ is not an integer multiple of d_i.*

Proof :
If part: We want to show that if $x_1 = x_2$ then $I_2 - I_1 = md_i$ for some integer m. Assume that I_0 is the first index in which token x_1 (x_2) is used (and is also regenerated). Then $I_0 \leq I_1$ and $I_0 \leq I_2$ because of lexicographical ordering. From the definition of the data-dependence vector, $I_1 = I_0 + m_1 d_i$ and $I_2 = I_0 + m_2 d_i$ for some non-negative integers m_1 and m_2. Therefore, $I_2 - I_1 = (m_2 - m_1)d_i = md_i$ for some integer m.

Only if part: We want to show that if $x_1 \neq x_2$ then $I_2 - I_1 \neq md_i$ for all integers m. Let $\hat{I}_1$ be the first index in which token x_1 is used (and is also regenerated) and let $\hat{I}_2$ be the first index in which token x_2 is used (and is also regenerated). Then x_1 will be used (and will be regenerated) in all indices of the form $\hat{I}_1 + \hat{m}_1 d_i \in I^p$ and x_2 will be used (and will be regenerated) in all indices of the form $\hat{I}_2 + \hat{m}_2 d_i \in I^p$.

Let us examine whether we could have $\hat{I}_2 = \hat{I}_1 + \hat{m}_1 d_i$ for some integer $\hat{m}_1$. If this equality holds, then both x_1 and x_2 are used in the index $\hat{I}_2 = \hat{I}_1 + \hat{m}_1 d_i$. However, as stated at the beginning of Section 4, at most one token of any data stream can be used in an index. Thus, $(\hat{I}_2 - \hat{I}_1) \neq \hat{m} d_i$ for all integers $\hat{m}$.

Then, from the definition of data-dependence vector, $I_1 = \hat{I}_1 + m_1 d_i$ and $I_2 = \hat{I}_2 + m_2 d_i$ for some non-negative integers m_1 and m_2. Therefore, $I_2 - I_1 = (\hat{I}_2 - \hat{I}_1) + (m_2 - m_1)d_i \neq md_i$ for all integers m. □

We will examine the case where $(I_2 - I_1) \neq md_i$ for all integers m. Then we will show that we cannot have $\mathbf{H}^1(I_2 - I_1)\mathrm{S}d_i = \mathrm{S}(I_2 - I_1)\mathbf{H}^1 d_i$, as otherwise collisions would occur in data links. As this is rather non-intuitive, we start with an example.

Let $\mathbf{H}^1 = (2,1,2)$ and $\mathrm{S} = (1,1,-2)$ for the matrix multiplication algorithm. This mapping $(\mathbf{H}^1, \mathrm{S})$ satisfies Conditions 1 through 4 and the resulting behavior is described in Fig-7. Observe that $C[0,3]$ collides with $C[2,0]$ and $C[1,3]$ collides with $C[3,0]$ because $\mathbf{H}^1((2,0,0)^t - (0,3,0)^t)\mathrm{S}(0,0,1)^t = \mathrm{S}((2,0,0)^t - (0,3,0)^t)\mathbf{H}^1(0,0,1)^t = -2$ and $(2,0,0)^t - (0,3,0)^t \neq m(0,0,1)^t$ as well as $\mathbf{H}^1((3,0,0)^t - (1,3,0)^t)\mathrm{S}(0,0,1)^t = \mathrm{S}((3,0,0)^t - (1,3,0)^t)\mathbf{H}^1(0,0,1)^t = -2$ and $(3,0,0)^t - (1,3,0)^t \neq m(0,0,1)^t$ for any m. To prevent this, we have the condition:

Condition 5 : *If* $(I_2 - I_1) \neq md_i$ *for all integers* m, *then* $\mathbf{H}^1(I_2 - I_1)\mathrm{S}d_i \neq \mathrm{S}(I_2 - I_1)\mathbf{H}^1 d_i$.

Lemma 8 : *Condition 5 is necessary.*

Proof : We consider three cases:

1. $\mathbf{H}^1 I_1 = \mathbf{H}^1 I_2$.

From Condition 2, it follows that $\mathbf{S}I_2 \neq \mathbf{S}I_1$. $\mathbf{H}^1(I_2 - I_1)\mathbf{S}d_i = (\mathbf{H}^1 I_2 - \mathbf{H}^1 I_1)\mathbf{S}d_i = 0$; however, from Condition 1, $\mathbf{H}^1 d_i > 0$ and therefore $\mathbf{S}(I_2 - I_1)\mathbf{H}^1 d_i = (\mathbf{S}I_2 - \mathbf{S}I_1)\mathbf{H}^1 d_i \neq 0$.

2. $\mathbf{S}I_1 = \mathbf{S}I_2$.
From Condition 2, it follows that $\mathbf{H}^1 I_2 \neq \mathbf{H}^1 I_1$. As we assume that $\mathbf{S}d_i \neq 0$, $\mathbf{H}^1(I_2 - I_1)\mathbf{S}d_i \neq 0$. However, as $\mathbf{S}I_2 = \mathbf{S}I_1$, $\mathbf{S}(I_2 - I_1)\mathbf{H}^1 d_i = 0$.

3. $\mathbf{H}^1 I_1 \neq \mathbf{H}^1 I_2$ and $\mathbf{S}I_1 \neq \mathbf{S}I_2$.
Let x_1 and x_2 be two tokens of the variable X corresponding to d_i, such that x_1 is the token used in I_1 and x_2 is the token used in I_2. Consider two subcases:

(a) d_i is a type 1 data-dependence vector.
Since $I_2 - I_1 \neq md_i$ for all integers m, from Lemma 7, $x_1 \neq x_2$. In addition, x_1 is in $PE_{\mathbf{S}I_1}$ at time $\mathbf{H}^1 I_1$ and x_2 is in $PE_{\mathbf{S}I_2}$ at time $\mathbf{H}^1 I_2$.
Without loss of generality, let $\mathbf{H}^1 I_1 < \mathbf{H}^1 I_2$. Let PE_b be the PE in which x_1 is located at time $\mathbf{H}^1 I_2$. Then by Lemma 3, $(\mathbf{H}^1 I_2 - \mathbf{H}^1 I_1)\mathbf{S}d_i = (b - \mathbf{S}I_1)\mathbf{H}^1 d_i$. Assume by contradiction that $\mathbf{H}^1(I_2 - I_1)\mathbf{S}d_i = \mathbf{S}(I_2 - I_1)\mathbf{H}^1 d_i$. Then, $\mathbf{S}(I_2 - I_1)\mathbf{H}^1 d_i = (b - \mathbf{S}I_1)\mathbf{H}^1 d_i$. From here, using $\mathbf{H}^1 d_i \neq 0$, $b = \mathbf{S}I_2$. Thus, the tokens x_1 and x_2 both use data link i and appear in $PE_{\mathbf{S}I_2}$ at time $\mathbf{H}^1 I_2$. It is "data collision," which is not allowed. Therefore, $\mathbf{H}^1(I_2 - I_1)\mathbf{S}d_i \neq \mathbf{S}(I_2 - I_1)\mathbf{H}^1 d_i$.

(b) d_i is a type 2 data-dependence vector.
Let x_k be a token of d_i generated in $\bar{I}_k - d_i \in I^p$ and used in $\bar{I}_k \in I^p$. It is used once only and therefore its value can be destroyed afterwards. However, we will view x_k in a natural way as being a member of a certain sequence of tokens. More specifically, let $j_{x_k} = \max\{j | \bar{I}_k - jd_i \in I^p\}$, $\bar{I}_k^{first} = \bar{I}_k - j_{x_k} d_i$, and let $\bar{I}_k^{last}$ be the last index in I^p of the form $\bar{I}_k^{first} + md_i$.
There is a sequence of tokens naturally associated with the sequence of indices $\bar{I}_k^{first}$, $\bar{I}_k^{first} + d_i, \ldots, \bar{I}_k^{last}$. We denote this sequence by $\vec{x_k}$. Observe that the $\vec{x_k}$ is analogous to a single token of a data stream with type 1 data-dependence vector. Perform the above for our tokens x_1 and x_2 obtaining sequences of tokens $\vec{x_1}$ and $\vec{x_2}$. The rest of the proof proceeds similarly to the second paragraph of the case 3(a) replacing x_1 there by our $\vec{x_1}$ and replacing x_2 there by our $\vec{x_2}$.

□

An immediate Corollary, which we will sometimes find useful, is:

Condition 5' : *If* $d_i \neq md_j$ *and* $d_j \neq md_i$ *for all integers* m, *then* $\frac{\mathbf{H^1}d_i}{\mathbf{S}d_i} \neq \frac{\mathbf{H^1}d_j}{\mathbf{S}d_j}$.

We summarize the preceding discussion in:

Theorem 9 : *A linear-array algorithm* $(\mathbf{H^1}, \mathbf{S})$ *that maps correctly a p nested loop algorithm Ag into a linear array Ar must satisfy Condition 1 through Condition 5.* □

4.2 Sufficient Conditions

By assuming Condition 1 through Condition 5, we will show that $(\mathbf{H^1}, \mathbf{S})$ will preserve the data-dependence relation, the right tokens will be in the right place at the right time, and in addition, no data tokens will collide in data links. Therefore, Condition 1 through Condition 5 are not only necessary, but also sufficient.

Theorem 10 : *A linear-array algorithm* $(\mathbf{H^1}, \mathbf{S})$ *from a p nested loop algorithm that satisfies Condition 1 through Condition 5 maps that p nested loop algorithm Ag correctly into a linear array Ar.*

Proof : Formally, the PEs in the array should be numbered from 1 to M, and $\mathbf{S}$ should map the indices into $\{PE_1, \ldots, PE_M\}$. However, it will be convenient to continue numbering the PEs from $\min\{\mathbf{S}I_1 | I_1 \in I^p\}$ to $\max\{\mathbf{S}I_2 | I_2 \in I^p\}$ in this proof.

First, from Condition 1, $\mathbf{H^1}$ preserves the data-dependence ordering.

Second, we show that the "needed" tokens will flow to the right place at the right time, in addition, $F_{Ar} = \hat{F}_{Ag}$ ("$\hat{F}_{Ag}$" was introduced at the beginning of Section 4). Because there are $K = w$ data-dependence vectors, we can let $\hat{F}_{Ag}$ be the K-ary functions executed at each loop index $\bar{I}$ of I^p. We will only consider assignment statements, as other types of statements can be handled by trivial modifications. Consider then, a typical assignment statement $r := \hat{F}_{Ag}(v_1, v_2, \ldots, v_K)$, where v_i is the token (variable) with the data-dependence vector d_i which was generated in $\bar{I}_i$ ($\bar{I}_i \in I^p$). We will now show by induction on $\mathbf{H^1}\bar{I}$ that all tokens v_i arriving at $PE_{\bar{I}}$ at time $\mathbf{H^1}\bar{I}$ will have correct values.

Basic step: When $\mathbf{H^1}\bar{I} = \min\{\mathbf{H^1}I_1 | I_1 \in I^p\}$, all tokens v_i are initial input/output tokens (variables). From Conditions 3 and 4 or Corollary 5, v_i will rearch $PE_{\mathbf{S}\bar{I}}$ at time $\mathbf{H^1}\bar{I}$. Thus, F_{Ar} performs the same function as $\hat{F}_{Ag}$ in $\bar{I}$.

Hypothesis step: Before the time $\mathbf{H^1}\bar{I}$, suppose that under $(\mathbf{H^1}, \mathbf{S})$ F_{Ar} has generated the same values as $\hat{F}_{Ag}$.

Induction step: At time $\mathbf{H^1}\bar{I}$, from the definition of the data-dependence vectors we have

$$\bar{I} = \bar{I}_1 + d_1 = \bar{I}_2 + d_2 = \ldots = \bar{I}_K + d_K, \tag{1}$$

for appropriate $\bar{I}_1, \bar{I}_2, \ldots, \bar{I}_K$. From the hypothesis step, v_i was regenerated with correct value in $PE_{\mathbf{S}\bar{I}_i}$ at time $\mathbf{H}^1\bar{I}_i$. However, v_i was not used (and was not modified) between the time instances $\mathbf{H}^1\bar{I}_i$ and $\mathbf{H}^1(\bar{I}_i + d_i)$ (not including $\mathbf{H}^1\bar{I}_i$ and $\mathbf{H}^1(\bar{I}_i + d_i)$). Then from Conditions 3 and 4 or Corollary 4, v_i will reach $PE_{\mathbf{S}\bar{I}_i+\mathbf{S}d_i}$ at time $\mathbf{H}^1\bar{I}_i + \mathbf{H}^1 d_i$. But from (1), $\mathbf{S}\bar{I} = \mathbf{S}(\bar{I}_i + d_i)$ and $\mathbf{H}^1\bar{I} = \mathbf{H}^1(\bar{I}_i + d_i)$; therefore, all of the tokens $v_1, v_2, \ldots, v_K$ arriving at $PE_{\mathbf{S}\bar{I}}$ at time $\mathbf{H}^1\bar{I}$ will have correct values. F_{Ar} will thus perform the same function as $\hat{F}_{Ag}$ in $\bar{I}$.

Third, we will show that no two tokens (variables) collide in any data link. Assume by contradiction that data link i has two distinct tokens x_1 and x_2 of the variable X which collide in PE_a during the execution. (Thus, of course, d_i is with X.) Suppose then that x_1 is then immediately used in index I_1 in $PE_{\mathbf{S}I_1}$ at time $\mathbf{H}^1 I_1$ and x_2 is then immediately used in index I_2 in $PE_{\mathbf{S}I_2}$ at time $\mathbf{H}^1 I_2$. (If x_1 or x_2 will not be used again, then x_1 was just generated in index I_1 or x_2 was just generated in index I_2.) Consider two cases:

1. $\mathbf{H}^1 I_1 = \mathbf{H}^1 I_2$.

   Our assumption is that at some time instance x_1 and x_2 collide in PE_a, and then x_1 is in $PE_{\mathbf{S}I_1}$ at time $\mathbf{H}^1 I_1 = \mathbf{H}^1 I_2$ and x_2 is in $PE_{\mathbf{S}I_2}$ at time $\mathbf{H}^1 I_2 = \mathbf{H}^1 I_1$. However, since all tokens of a data stream flow at the same speed, we have $PE_{\mathbf{S}I_1} = PE_{\mathbf{S}I_2}$ contradicting Condition 2.

2. $\mathbf{H}^1 I_1 \neq \mathbf{H}^1 I_2$.

   Without loss of generality, $\mathbf{H}^1 I_1 < \mathbf{H}^1 I_2$. Since x_1 and x_2 flow at the same speed, x_2 will flow from PE_a to $PE_{\mathbf{S}I_1}$ at time $\mathbf{H}^1 I_1$ and then to $PE_{\mathbf{S}I_2}$ at time $\mathbf{H}^1 I_2$. As all tokens flow with non-zero velocity, $\mathbf{S}I_1 \neq \mathbf{S}I_2$. Consider two subcases:

   (a) $I_2 - I_1 \neq m d_i$ for all integers m.

   From Lemma 3, token x_2 is in $PE_{\mathbf{S}I_1}$ at time $\mathbf{H}^1 I_1$, is in $PE_{\mathbf{S}I_2}$ at time $\mathbf{H}^1 I_2$, and therefore $\frac{\mathbf{H}^1 I_2 - \mathbf{H}^1 I_1}{\mathbf{S}I_2 - \mathbf{S}I_1} = \frac{\mathbf{H}^1 d_i}{\mathbf{S}d_i}$. However, this contradicts Condition 5.

   (b) $I_2 - I_1 = m d_i$ for some integer m.

   Consider two cases:

   i. d_i is a type 1 data-dependence vector.

   From Lemma 7, $x_1 = x_2$. However, this contradicts our assumption that $x_1 \neq x_2$.

   ii. d_i is a type 2 data-dependence vector.

   Since x_2 is used in I_2, x_2 is generated only in $I_2 - d_i$. However, since x_2 was in $PE_{\mathbf{S}I_1}$ and $I_2 \neq I_1$ and $I_2 - I_1 = m d_i$ for some integer m, $I_1 = I_2 - d_i$ and $PE_a = PE_{\mathbf{S}I_1}$. From the definition of type 2 data-dependence vector, x_1 will not be used again after it is used in I_1. Therefore, when x_2 is generated in $PE_{\mathbf{S}(I_2 - d_i)} = PE_{\mathbf{S}I_1}$, x_1 is destroyed. Thus, no collisions will occur.

Finally, from Theorem 1, $(\mathbf{H^1}, \mathbf{S})$ can be performed in parallel. □

4.3 Storage and Time Complexity

In this subsection, we study the storage complexity and the time complexity of the mapping $(\mathbf{H^1}, \mathbf{S})$.

Theorem 11 :

1. *Total number of PEs is* $M = \max\{|\mathbf{S}(I_2 - I_1)| \mid I_1, I_2 \in I^p\} + 1$.
2. *Total execution time is* $T = O(\max\{|\mathbf{H^1}(I_2 - I_1)| \mid I_1, I_2 \in I^p\} + N)$[1], *where* $N = M(1 + \sum_{i=1}^{K} b_i)$.

Proof :

1. Follows immediately from the structure of the mapping.
2. The time complexity, T, of the linear array implementation is the time elapsed between the instance the first token is input into the array and the instance the last token is output from the array. Let T' be the time elapsed between the instance the first index is executed and the instance the last index is executed. Both the time between the instance the first token is input into the array and the instance the first index is executed and the time between the instance the last index is executed and the instance the last token is output from the array are less equal to $M(\max\{b_i | 1 \leq i \leq K\} + 1)$. Thus $T \leq T' + 2M(\max\{b_i | 1 \leq i \leq K\} + 1)$.

   As $T' = \max\{|\mathbf{H^1}(I_2 - I_1)| \mid I_1, I_2 \in I^p\}$ and $M(\max\{b_i | 1 \leq i \leq K\} + 1) \leq N$, the proof follows immediately. □

Until now, in the linear array, PEs were numbered from $\min\{\mathbf{S}I_1 | I_1 \in I^p\}$ to $\max\{\mathbf{S}I_2 | I_2 \in I^p\}$. Using M from Theorem 11, we can renumber PEs from 1, 2, ..., M as is formally required. We will say that $(\mathbf{H^1}, \mathbf{S})$ uses an N-*storage* linear array, where $N = M(1 + \sum_{i=1}^{K} b_i)$, because N is the sum of the total number of the PEs and the local registers.

Corollary 12 : *Total execution time is* $T = \Omega(N)$.

[1] Let f and g be two functions of argument x. $f = O(g)$ if $f \leq cg$ for some constant c and for sufficiently large x. $f = \Omega(g)$ if $f \geq cg$ for some constant c and for sufficiently large x. $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$. $f = o(g)$ if $f/g \to 0$ as $x \to \infty$.

Proof : Since there is at least one data token passing through data link i for $1 \leq i \leq K$ and since K is independent of the problem size n and

$$M(\max\{b_i|1 \leq i \leq K\} + 1) \leq N \leq KM(\max\{b_i|1 \leq i \leq K\} + 1),$$

the proof follows immediately. □

We now want to consider the case, where the set of the data-dependence vectors is $\{d_1, d_2, \ldots, d_w\} = \{(1,0,\ldots,0)^t, (0,1,\ldots,0)^t, \ldots, (0,0,\ldots,1)^t\}$. As mentioned above such algorithms include algorithms to solve matrix multiplication, L-U decomposition, inversion of nonsingularly triangular matrix [9], matrix orthogonal triangularization, a version of transitive closure algorithm [7], DFT, bubble sort [21], and others.

We are interested in tight bounds on time complexity of linear array implementations of these algorithms.

Theorem 13 : *If* $\{d_1, d_2, \ldots, d_w\} = \{(1,0,\ldots,0)^t, (0,1,\ldots,0)^t, \ldots, (0,0,\ldots,1)^t\}$ *is the set of data-dependence vectors in* D_{Ag}, *then the execution time is* $T = \Theta(N)$.

Proof : There are exactly p distinct data-dependence vectors in the set $\{d_1, d_2, \ldots, d_w\}$, so we assume, without loss of generality, that $K = w = p$ and $d_1 = (1,0,\ldots,0)^t$, $d_2 = (0,1,\ldots,0)^t$, $\ldots$, and $d_p = (0,0,\ldots,1)^t$. From Condition 1, $(l_1, l_2, \ldots, l_p)^t$ must be the first index to be executed and $(u_1, u_2, \ldots, u_p)^t$ must be the last index to be executed, because $\mathbf{H^1}d_1 > 0$, $\mathbf{H^1}d_2 > 0$, $\ldots$, and $\mathbf{H^1}d_p > 0$. In addition, $(u_1, u_2, \ldots, u_p)^t - (l_1, l_2, \ldots, l_p)^t = (u_1 - l_1)d_1 + (u_2 - l_2)d_2 + \cdots + (u_p - l_p)d_p$. Since both $(l_1, l_2, \ldots, l_p)^t$ and $(l_1, l_2, \ldots, l_p)^t + (u_i - l_i)d_i$ will be mapped into the linear array, $(u_i - l_i)|\mathbf{S}d_i| \leq M$. Let $T' = \mathbf{H^1}(u_1, u_2, \ldots, u_p)^t - \mathbf{H^1}(l_1, l_2, \ldots, l_p)^t$. then,

$$\begin{array}{rcll} T' & = & \sum_{i=1}^{K}(u_i - l_i)\mathbf{H^1}d_i & \text{(because } K = p\text{)} \\ & = & \sum_{i=1}^{K}(u_i - l_i)|\mathbf{S}d_i|(b_i + 1) & \text{(from Lemma 3)} \\ & \leq & \sum_{i=1}^{K} M(b_i + 1) & \text{(because } (u_i - l_i)|\mathbf{S}d_i| \leq M\text{)} \\ & \leq & M(\sum_{i=1}^{K} b_i + K). & \end{array}$$

Since K is independent of the problem size n, $T' = O(N)$. In addition, $T \leq T' + 1 + 2M(\max\{b_i|1 \leq i \leq K\} + 1) = O(N)$. Finally, from Corollary 12 we have $T = \Theta(N)$. □

Theorem 13 shows that the tight bound on time complexity of this class of algorithms is the same as the storage complexity. We can use this rather general result to show that some mappings have both the optimal time complexity and optimal storage complexity.

As an example, consider the $n \times n$ matrix multiplication algorithm $Ag = (\{(i,j,k)^t | 0 \leq i,j,k, \leq n-1\}, \{A,B,C\}, (C[i,j] := C[i,j] + A[i,k] * B[k,j]))$. In Section 3 (and also in [31]), a linear array algorithm with $\mathbf{H^1} = (2,1,n-1)$ and $\mathbf{S} = (1,1,-1)$ was synthesized for this algorithm. (Actually, Section 3 dealt with $n = 4$, but the extension to general n is trivial.) In this mapping, the total number of PEs is $(1,1,-1)((n-1,n-1,0)^t - (0,0,n-1)^t) = 3n-2$. In addition, there is one register for A ($b_1 = 1$), no registers for B ($b_2 = 0$), and $n-2$ registers for C ($b_3 = n-2$). Thus, the total storage complexity of the linear array is $N = (3n-2)\sum_{i=1}^{3}(b_i+1) = (3n-2)(2+1+n-1)$ $= \Theta(n^2)$. As by Theorem 13, $T = \Theta(N)$, we see that $T = \Theta(n^2)$.

We now show that this linear array is time optimal. As there are $3n^2$ tokens to be read and there is only a constant number (three) of Input ports, the input time required is $\Omega(n^2)$, and thus the algorithm's time complexity is optimal.

We will now show that the storage complexity must be $\Omega(n^2)$, thus proving that the array is storage optimal. Assume by contradiction that for some array implementing matrix multiplication $N = o(n^2)$. Then by Theorem 13, $T = o(n^2)$ contradicting the fact that $T = \Omega(n^2)$. Thus the algorithm above was optimal. Note: Ramakrishnan and Varman [32] also obtained this bound by reducing matrix multiplication to a game played with tokens on an undirected graph.

As we will see later, all the linear array algorithms for matrix multiplication in Sections 5 and 6 have both the optimal time complexity and the optimal storage complexity. The storage can be distributed in between $O(n)$ and $O(n^2)$ PEs depending on additional objectives of the designer.

5 The trade-offs

In this section, we consider the trade-offs between the time, the number of PEs and the number of local registers in each PE.

In Subsection 3.3, we showed how to find the time hyperplane $\mathbf{H^1} = (2,1,3)$ when given a space mapping $\mathbf{S} = (1,1,-1)$ for matrix multiplication. Furthermore, in Section 4 we specified conditions to be satisfied in deriving a time hyperplane $\mathbf{H^1}$ for a given space mapping $\mathbf{S}$. We now want to consider explicitly the case where we are restricted to designing the linear array with a specific type of PE provided to us in order to avoid custom design of the PEs. Alternatively, for fabrication reasons, we may be restricted to designing PEs with l data links of positive direction and r data links of negative direction for specific l and r. To further elaborate, it is of importance to consider such restrictions because we may have only one type of PE available, or for reasons of mass production we wish to produce only a small number of types of PEs but still should be able to use them to design arrays solving a variety of problems. Such restrictions give rise to additional constraints on the design which we will now consider.

We continue with the example of matrix multiplication. Earlier, we derived an implementation

of the matrix multiplication algorithm characterized by $\mathbf{H^1} = (2,1,3)$ and $\mathbf{S} = (1,1,-1)$. We see that:

there is no shifting register for A because $|\frac{\mathbf{H^1}(0,1,0)^t}{\mathbf{S}(0,1,0)^t}| - 1 = 0$,
there is one : B because $|\frac{\mathbf{H^1}(1,0,0)^t}{\mathbf{S}(1,0,0)^t}| - 1 = 1$,
there are two shifting registers for C because $|\frac{\mathbf{H^1}(0,0,1)^t}{\mathbf{S}(0,0,1)^t}| - 1 = 2$.

We now discuss how to utilize PEs in which the values of the parameters b_i's and t_i's are not under our control, and we have to design an array within these parameters. From Lemma 3, the equation

$$\frac{\mathbf{H^1} d_i}{\mathbf{S} d_i} = t_i(b_i + 1) \tag{2}$$

must be satisfied for each i.

Let link 1 correspond to A, link 2 correspond to B, and link 3 correspond to C. Assume now that we are provided with a PE in which $b_1 = 0$, $b_2 = 1$, $b_3 = 1$, and $t_1 = 1$, $t_2 = 1$, $t_3 = -1$. From (2), the following must be satisfied:

$$\frac{\mathbf{H_2^1}(0,1,0)^t}{\mathbf{S_2}(0,1,0)^t} = 1, \frac{\mathbf{H_2^1}(1,0,0)^t}{\mathbf{S_2}(1,0,0)^t} = 2, \text{ and } \frac{\mathbf{H_2^1}(0,0,1)^t}{\mathbf{S_2}(0,0,1)^t} = -2.$$

Let $\mathbf{H_2^1} = (2\delta, 1, 2\tau)$ and $\mathbf{S_2} = (\delta, 1, -\tau)$. From Condition 1, $\delta, \tau > 0$, from Condition 2, $2\delta x + y + 2\tau z = 0$ and $\delta x + y - \tau z = 0$ can not both be true at the same time, where $-(n-1) \leq x, y, z \leq n-1$. For $n = 4$, we have $-3 \leq x, y, z \leq 3$. Finally, from Condition 5, we obtain the following three inequalities: 1. $2\delta x + y + 2\tau z \neq \delta x + y - \tau z$ (unless $(x,y,z)^t = md_1$ for some m), 2. $2\delta x + y + 2\tau z \neq 2\delta x + 2y - 2\tau z$ (unless $(x,y,z)^t = md_2$ for some m), 3. $2\delta x + y + 2\tau z \neq -2\delta x - 2y + 2\tau z$ (unless $(x,y,z)^t = md_3$ for some m). We can let $\delta = 1$ and $\tau = 2$. Then, $\mathbf{H_2^1} = (2,1,4)$ and $\mathbf{S_2} = (1,1,-2)$ satisfy Theorem 10 and the resulting mapping is shown in Fig-8.

Similarly, assume that we are provided with a PE in which $b_1 = 0$, $b_2 = 1$, $b_3 = 0$, and $t_1 = 1$, $t_2 = 1$, $t_3 = -1$. From (2), the following must be satisfied:

$$\frac{\mathbf{H_3^1}(0,1,0)^t}{\mathbf{S_3}(0,1,0)^t} = 1, \frac{\mathbf{H_3^1}(1,0,0)^t}{\mathbf{S_3}(1,0,0)^t} = 2, \text{ and } \frac{\mathbf{H_3^1}(0,0,1)^t}{\mathbf{S_3}(0,0,1)^t} = -1.$$

Let $\mathbf{H_3^1} = (2\delta, 1, \tau)$ and $\mathbf{S_3} = (\delta, 1, -\tau)$. From Condition 1, $\delta, \tau > 0$, from Condition 2, $2\delta x + y + \tau z = 0$ and $\delta x + y - \tau z = 0$ can not both be true at the same time, where $-(n-1) \leq x, y, z \leq n-1$. For $n = 4$, we have $-3 \leq x, y, z \leq 3$. Finally, from Condition 5, we obtain the following three inequalities: 1. $2\delta x + y + \tau z \neq \delta x + y - \tau z$ (unless $(x,y,z)^t = md_1$ for some m), 2. $2\delta x + y + \tau z \neq 2\delta x + 2y - 2\tau z$ (unless $(x,y,z)^t = md_2$ for some m), 3. $2\delta x + y + \tau z \neq -\delta x - y + \tau z$ (unless $(x,y,z)^t = md_3$ for some m). We can let $\delta = 3$ and $\tau = 2$. Then, $\mathbf{H_3^1} = (6,1,2)$ and $\mathbf{S_3} = (3,1,-2)$ satisfy Theorem

10 and the resulting mapping is shown in Fig-9.

As the reader can see, the above three examples (the original one and the two new ones) share the parameters $b_1 = 0$, $b_2 = 1$, and $t_1 = 1$, $t_2 = 1$, $t_3 = -1$. Thus, they are all special cases of a family of solutions defined by

$$\frac{\mathbf{H_4^1}(0,1,0)^t}{\mathbf{S_4}(0,1,0)^t} = 1, \frac{\mathbf{H_4^1}(1,0,0)^t}{\mathbf{S_4}(1,0,0)^t} = 2, \text{ and } \frac{\mathbf{H_4^1}(0,0,1)^t}{\mathbf{S_4}(0,0,1)^t} = -(b_3+1). \quad (3)$$

Families of solutions were studied before, for instance by Varman and Ramakrishnan [40]. We can show that the family studied by them is a special case of the family defined by (3). Specifically, let us further restrict (3) by imposing the condition that the only free parameter $b_3 + 2$ in (3) is restricted being prime. Using our notation, we can say that they found that if $b_3 + 2$ is prime then mapping the index $(i,j,k)^t$ to $PE_{\delta \times i + j - (\delta+1) \times k}$ at time $2 \times \delta \times i + j + (\delta+1) \times (b_3+1) \times k$ is a family of solutions, where $\hat{\delta} = \lceil \frac{n}{b_3+2} \rceil$, and $\delta = \hat{\delta} + 1$ if $b_3 + 2$ divides $\hat{\delta}$, otherwise $\delta = \hat{\delta}$. Their solutions can be obtained by specifying $\mathbf{H}^{1'} = (2\delta, 1, (\delta+1)(b_3+1))$ and $\mathbf{S}' = (\delta, 1, -(\delta+1))$.

However, (3) admits other solutions too, for instance the two solutions above: $(\mathbf{H_2^1}, \mathbf{S_2})$ and $(\mathbf{H_3^1}, \mathbf{S_3})$. Recall, that formula (2) is even more general than formula (3) and thus we can find solutions satisfying formula (2) that do not belong to the family defined by formula (3).

Our method is also sufficiently general to encompass some additional results obtained by other researchers previously. We now list them for completeness:

1. If we let $\mathbf{H_a^1} = (2,1,n)$ and $\mathbf{S_a} = (1,1,0)$, then we obtain the algorithms in Kulkarni and Yen [13] and Kung [17].

   Note: in this case, the tokens of C are fixed in the PEs.

2. If we let $\mathbf{H_b^1} = (2,1,n-1)$ and $\mathbf{S_b} = (1,1,-1)$, then we obtain the algorithm in Ramakrishnan *et al.* [31].

3. If we let $\mathbf{H_c^1} = (2\delta,1,\tau)$ and $\mathbf{S_c} = (\delta,1,-\tau)$ for $\delta = n$ and $\tau = \frac{n+1}{2}$ when n is odd, and for $\delta = n-1$ and $\tau = \frac{n}{2}$ when n is even, then we obtain the algorithms in Ramakrishnan and Varman [32].

4. $(\mathbf{H}^{1'}, \mathbf{S}')$ is the "Full-Pipeline" algorithm in Varman and Ramakrishnan [40].

5. If we let $\mathbf{H_d^1} = (2\delta,1,\nu)$ and $\mathbf{S_d} = (\delta,1,0)$ for prime ν, and $n < \nu < 2n$, and there are $\lceil \frac{n}{\delta} \rceil$ local registers for C in each PE, then we obtain the "Partial-Pipeline" algorithm in Varman and Ramakrishnan [40].

   Note: in this case, the tokens of C are fixed in the PEs.

The other objectives in designing the array may include the minimization of the execution time or the number of PEs used. To state formally:

1. If we want to obtain an algorithm with minimum execution time, then we add the constraint

$$\mathbf{H}^1 = \min_{\bar{\mathbf{H}}^1}\{\max\{|\bar{\mathbf{H}}^1(I_2 - I_1)| \mid I_1, I_2 \in I^p\}\}.$$

2. If we want to obtain an algorithm with minimum number of PEs, then we add the constraint

$$\mathbf{S} = \min_{\bar{\mathbf{S}}}\{\max\{|\bar{\mathbf{S}}(I_2 - I_1)| \mid I_1, I_2 \in I^p\}\}.$$

Note that, of course, it may not in general be possible to satisfy these two objectives simultaneously.

We will not consider in this paper other optimization criteria such as the minimization of $PE/time$ product or $PE/time^2$ product or the maximization of throughput or PEs' utilization. These types of optimization generally require smaller number of PEs, more local registers in a PE, and more I/O ports. It is possible to create designs taking into account these criteria; however, they will not be modularly extensible, storage optimal, or bounded I/O. In addition, if the number of local registers for each data link i is greater than $\frac{\mathbf{H}^1 d_i}{\mathbf{S} d_i} - 1$, this requires memory addressing and control hardware, because of not purely systolic nature of the computation.

6 The partition model.

Consider the case when we are given a specific type of PE suitable for designing the required linear array algorithm. As the size of the problem to be solved grows (e.g., the dimensions of the matrices to be multiplied increase), we will need more and more PEs to construct an array solving the problem, as the amount of storage needed increases. What should we do then if the number of PEs provided to us is limited? This problem was studied extensively too, see [7] [9] [28] [2], who provided *partition* methods to deal with it.

The goal of Guibas *et al.* [7] and Hwang *et al.* [9] is to solve the problem by using 2-D arrays with limited number of PEs. They concentrated on specific algorithms. Guibas, Kung and Thompson [7] studied partitioning for dynamic programming, matrix multiplication, and transitive closure. Hwang and Cheng [9] proposed a set of four primitive chips which when properly interconnected could solve the matrix multiplication, LU decomposition, inversion of nonsingular triangular matrices, and the solution of linear systems of equation. Moldovan and Fortes [28] considered a general model for partitioning. They divided the index space I^p into bands according to the time hyperplane, assigned an ordering to execute the bands, and then mapped each band to a fixed $(p-1)$-D array.

Annaratone *et al.* [2] gave three partitioning methods: input partitioning, output partitioning, and pipelining. These three methods were applied to the WARP computer. Input partitioning and output partitioning are not systolic. Pipelining is systolic; however, it is a partial pipeline (some data are fixed in PEs and some data are pipelined). Partial pipelining algorithms were partitioned by assigning each row or column of a matrix to a PE, and each PE then performs one stage of the processing. The intermediate result of one PE will be sent to the next PE resulting in pipeline. Since WARP computer has only a 1-D WARP array with ten PEs, the algorithm requires many scans for a 512 × 512 matrix multiplication problem.

We will also study partitioning of algorithms. Suppose, $I^p = \{(i_1, i_2, \ldots, i_p)^t \mid l_j \leq i_j \leq u_j \text{ and } u_j - l_j = O(n) \text{ for } j = 1, 2, \ldots, p\}$. For each value of n, we require a linear array of some storage size. If we are given an array with smaller storage, we need to partition the algorithm. We will show how to do that if all the data streams flow in the same direction. In the sequel we will refer to the algorithms considered so far as *unpartitioned.*

Suppose then that for problem size characterized by n as above, we need an $N = N(n)$ storage linear array and requiring T time for an unpartitioned algorithm. If we are given only a k-storage linear array for $k < N$, then we will consider feeding the data streams $\lceil N/k \rceil$ times into this k-storage linear array. This approach is illustrated in Fig-10. In Fig 10-a, if the data streams need to be fed into the N-storage linear array once only, then in Fig 10-b, the data streams will be fed into the k-storage linear array $m = \lceil N/k \rceil$ times. Thus, the new algorithm can be naturally divided into m *phases.*

Since T time is required to feed the data streams into the linear array once, the total execution time is $O(TN/k)$. Next, suppose that this k-storage linear array contains q PEs and some index $\bar{I}$ is executed in $PE_{\mathbf{S}\bar{I}}$ in the unpartitioned algorithm. Then, $\bar{I}$ will be executed in $PE_{((\mathbf{S}\bar{I} - \min\{\mathbf{S}I_1 | I_1 \in I^p\} + 1) \bmod q)}$ in the partitioned algorithm ($a \bmod b \in \{1, 2, \ldots, b\}$). Furthermore, it is executed at time $\mathbf{H}^1\bar{I}$ in phase $\lceil (\mathbf{S}\bar{I} - \min\{\mathbf{S}I_1 | I_1 \in I^p\} + 1)/q \rceil$. We state the above as the following condition:

Condition 6: *If all the data streams have the same direction, i.e. for all* d_i, $\mathbf{S}d_i > 0$, *and* q *is the number of PEs in the available linear array, then for the partition algorithm* $(\mathbf{H}^1_{\mathbf{q}}, \mathbf{S}_{\mathbf{q}})$*:*

- $\mathbf{H}^1_{\mathbf{q}}\bar{I} = \mathbf{H}^1\bar{I}$ in phase $\lceil (\mathbf{S}\bar{I} - \min\{\mathbf{S}I_1 | I_1 \in I^p\} + 1)/q \rceil$, and
- $\mathbf{S}_{\mathbf{q}}\bar{I} = (\mathbf{S}\bar{I} - \min\{\mathbf{S}I_1 | I_1 \in I^p\} + 1) \bmod q$.

Note: This condition applies if we do not utilize any information about special properties of the specific problem and its unpartitioned solution that could hold for some special cases. Thus, it is applicable to general p nested loop algorithms. We are studying some important problems for

which advantage can be taken of the problem's structure. They are, however, beyond the scope of this paper.

As an example, we will now present a family of unpartitioned matrix-multiplication algorithms, which can be easily partitioned.

As before, let link 1 correspond to A, link 2 correspond to B, and link 3 correspond to C. Suppose that we are restricted to PEs characterized by $b_1 = 0$, $b_1 = 1$, and $t_1 = t_2 = t_3 = 1$. We want to derive conditions on b_3. From Condition 6, $\mathbf{S}d_i > 0$, from Condition 1, $\mathbf{H^1}d_i > 0$, and from Conditions 3 and 4, $\frac{\mathbf{H^1}d_i}{\mathbf{S}d_i} = b_3 + 1$. Therefore, we must choose $\mathbf{H^1} = (2\delta, 1, \tau(b_3+1))$ and $\mathbf{S} = (\delta, 1, \tau)$, where $\delta > 0$ and $\tau > 0$. Next, from Condition 5', $\frac{2\delta}{\delta} \neq \frac{1}{1}$ and $\frac{1}{1} \neq \frac{\tau(b_3+1)}{\tau}$ and $\frac{\tau(b_3+1)}{\tau} \neq \frac{2\delta}{\delta}$, implying $(b_3+1) > 2$, or $b_3 \geq 2$. Then, from Condition 2, $2\delta x + y + \tau(b_3+1)z = 0$ and $\delta x + y + \tau z = 0$ can not both be true at the same time (where $-(n-1) \leq x, y, z \leq n-1$). Finally, from Condition 5, we obtain the following three inequalities:

$$-y + \tau(b_3 - 1)z \neq 0 \quad (\text{unless } (x,y,z)^t = md_1 \text{ for some } m) \tag{4}$$

$$\delta x + \tau b_3 z \neq 0 \quad (\text{unless } (x,y,z)^t = md_2 \text{ for some } m) \tag{5}$$

$$\delta(b_3 - 1)x + b_3 y \neq 0 \quad (\text{unless } (x,y,z)^t = md_3 \text{ for some } m) \tag{6}$$

To find simple solutions for δ and τ so that (4)-(6) are satisfied, we can require [2]

$$\tau(b_3 - 1) \geq n \gcd(\tau(b_3 - 1), 1) \tag{7}$$

$$\tau b_3 \geq n \gcd(\tau b_3, \delta) \tag{8}$$

$$\delta(b_3 - 1) \geq n \gcd(\delta(b_3 - 1), b_3) \tag{9}$$

In particular, if b_3 is prime, then first let $\hat{\delta} = \lceil \frac{n}{b_3 - 1} \rceil$. Second, if b_3 divides $\hat{\delta}$, then let $\delta = \hat{\delta} + 1$ and $\tau = \hat{\delta}$; otherwise let $\delta = \hat{\delta}$ and $\tau = \delta + 1$.

Such a choice of δ and τ will satisfy (7), (8), and (9), because $\gcd(\tau b_3, \delta) = \gcd(\tau(b_3-1), 1) = \gcd(\delta(b_3-1), b_3) = 1$ and $\tau b_3, \tau(b_3-1), \delta(b_3-1) \geq n$. Therefore, based on this choice, $(\mathbf{H^1}, \mathbf{S})$ is a family of algorithms for which all data streams flow in the same direction. In addition, both the time complexity and the total size of the storage including the number of PEs and the number of the local registers in each PE are $O(n^2)$. To create the partitioned version of these algorithms is straightforward and we ommit the details from this presentation.

[2] Varman and Ramakrishnan [40] showed that if x, y, n, α and β are integers such that $n > 0, 0 < |x|, |y| < n$, $1 \leq \alpha \leq n$, $\beta \geq n \gcd(\alpha, \beta)$, then $\alpha x \neq \beta y$.

7 Two Path-Finding Algorithms

Path-finding problems [1] include the problems to compute the transitive closure and all shortest paths. Kung [19] and Rote [34] also show that nonsingular matrix-inversion problem is a special case of the path-finding problem. For 2-D systolic arrays to solve path-finding problems, consult [7] [34] [23] [19].

Guibas, Kung and Thompson [7] first proposed a three-pass mesh-connected systolic-array algorithm for the path-finding problem. This algorithm is very important for pragmatical reasons, because, the data streams of each pass are the same as that of the matrix multiplication algorithm described in Section 3. Therefore, we can construct a three-pass linear-array algorithm for the path-finding problem based on their algorithm, which both is modularly extensible and provides a partitioned algorithm. Varman and Ramakrishnan [37] also gave a solution for this method.

However, in this section, we want to synthesize a linear-array algorithm directly from the Warshall-Floyd path-finding algorithm, because it is of independent interest for the following reasons: (1) this algorithm has five data-dependence vectors instead of three data-dependence vectors[3] in [7], (2) this linear-array algorithm needs only one pass of data streams in unpartitioned algorithm instead of three passes in the algorithm based on [7].

A. The Warshall-Floyd path-finding algorithm.

The transitive closure problem: Given an $n \times n$ Boolean matrix of elements $C[i,j]$ over an n node graph, with $C[i,j]$ being 1 if there is an arc from node i to node j or $i = j$, and 0, otherwise. Then, determine whether there is a path from node i to node j for all i, j.

The all-shortest-path problem: Given an $n \times n$ distance matrix of elements $D[i,j]$ over an n node graph, with $D[i,j]$ being the length of the arc from node i to node j, where $D[i,j] \geq 0$ and $D[i,i] = 0$ for all i, j. Then, determine what is the shortest distance from node i to node j for all i, j.

The transitive closure and the all shortest paths can be obtained by the Warshall-Floyd algorithm, where both input and solution will be in C:

for $k, i, j = 1$ to n
  $C[i,j] = C[i,j] \oplus (C[i,k] \otimes C[k,j])$
end_for

The above notation is a convenient representation for the 3 nested loop algorithm. $\oplus$ represents the logical $\vee$ (min) and $\otimes$ represents the logical $\wedge$ $(+)$ in the transitive-closure problem (all-shortest-path problem), respectively.

After adding indices to each variable, the algorithm becomes:

[3]Because there are three data-dependence vectors in matrix-multiplication algorithm.

for $k, i, j = 1$ to n
  if $j = 1$
    then $C^{(k,i,j)}[i,k] = C^{(k',i',j')}[i',j']$
    else $C^{(k,i,j)}[i,k] = C^{(k,i,j-1)}[i,k]$;
  if $i = 1$
    then $C^{(k,i,j)}[k,j] = C^{(k',i',j')}[i',j']$
    else $C^{(k,i,j)}[k,j] = C^{(k,i-1,j)}[k,j]$;
  $C^{(k,i,j)}[i,j] = C^{(k-1,i,j)}[i,j] \oplus (C^{(k,i,j)}[i,k] \otimes C^{(k,i,j)}[k,j])$
end_for

There are five data-dependence vectors:

$$\begin{array}{lll} d_1 = & (0,0,1)^t & \text{for } (C^{(k,i,j)}[i,k], C^{(k,i,j-1)}[i,k]) \\ d_2 = & (0,1,0)^t & \text{for } (C^{(k,i,j)}[k,j], C^{(k,i-1,j)}[k,j]) \\ d_3 = & (1,0,0)^t & \text{for } (C^{(k,i,j)}[i,j], C^{(k-1,i,j)}[i,j]) \\ d_4 = & (1,0,j-k)^t & \text{for } (C^{(k,i,j)}[i,k], C^{(k',i',j')}[i',j']) \\ d_5 = & (1,i-k,0)^t & \text{for } (C^{(k,i,j)}[k,j], C^{(k',i',j')}[i',j']) \end{array}$$

We have d_4, because $C[i,k] = C[i',j']$, $i = i'$, $k = j'$, and $k' = k - 1$, thus $(k,i,j)^t - (k',i',j')^t = (1,0,j-k)^t$. Similarly we have d_5, because $C[k,j] = C[i',j']$, $k = i'$, $j = j'$, and $k' = k - 1$, thus $(k,i,j)^t - (k',i',j')^t = (1,i-k,0)^t$. Although there is only one input stream of C, say $C[i,j]$ corresponding to d_3, during the execution it will create four data streams corresponding to d_1, d_2, d_4, and d_5, respectively. The data-dependence graph for $n = 3$ is shown in Fig-11. In this graph, it is easy to see that d_4 and d_5 are not constant. Since we cannot accommodate non-constant speed in a data stream, we have to fix the corresponding two data streams in PE's. If we let $\mathbf{S}_5 d_4 = \mathbf{S}_5 d_5 = 0$, it implies $\mathbf{S}_5 = (0,0,0)$, that is, all the indices will be mapped into PE_0. Therefore, we can not pipeline this algorithm in the linear array. Thus the algorithm must be modified.

B. A reindexed path-finding algorithm.

In transitive-closure problem, we have the following properties: First, there is always a path from node i to node i (i.e. initially, $C[i,i] = 1$), and second, $C^k[i,j] \geq C^l[i,j]$ if $k > l$, where $C^p[i,j] = 1$ means that there is a path from node i to node j passing through intermediate nodes from $\{1, 2, \ldots, p\}$ only, and $C^p[i,j] = 0$, otherwise. (Similarly in all-shortest-path problem: First $D[i,i] = 0$ because the shortest distance from node i to node i is zero, and second, $D^k[i,j] \leq D^l[i,j]$ if $k > l$, where $D^q[i,j]$ denotes the shortest distance from node i to node j passing through intermediate nodes in $\{1, 2, \ldots, q\}$ only.)

Now let us consider the following reindexed transitive-closure algorithm, which Kung *et al.* [19] have mapped to a 2-D spiral array and a 2-D orthogonal array:

for $k, i, j = 1$ to n
  $C[i + k - 1 \bmod n, j + k - 1 \bmod n] =$
    $C[i + k - 1 \bmod n, j + k - 1 \bmod n] \bigoplus$
    $(C[i + k - 1 \bmod n, k] \bigotimes C[k, j + k - 1 \bmod n])$
end_for /* Note: $i \bmod n \in \{1, 2, ..., n\}$ */

Based on the properties $C[i, i] = 1$ and $C^k[i, j] \geq C^l[i, j]$ for $k > l$, we can pipeline each variable and add the index to each variable to obtain:

for $k, i, j = 1$ to n
  if $i = n$
    if $j = n$
    then $C^{(k,n,n)}[n + k - 1 \bmod n, n + k - 1 \bmod n] = 1$
    else $C^{(k,n,j)}[n + k - 1 \bmod n, j + k - 1 \bmod n] =$
      $C^{(k',n,j')}[k', j' + k' - 1 \bmod n]$;
  else_if $j = n$
    then $C^{(k,i,n)}[i + k - 1 \bmod n, n + k - 1 \bmod n] =$
      $C^{(k',i',n)}[i' + k' - 1 \bmod n, k']$
    else $C^{(k,i,j)}[i + k - 1 \bmod n, j + k - 1 \bmod n] =$
      $C^{(k',i',j')}[i' + k' - 1 \bmod n, j' + k' - 1 \bmod n]$;
  if $j = 1$
    then $C^{(k,i,1)}[i + k - 1 \bmod n, k] =$
      $C^{(k,i,1)}[i + k - 1 \bmod n, 1 + k - 1 \bmod n]$
    else $C^{(k,i,j)}[i + k - 1 \bmod n, k] =$
      $C^{(k,i,j-1)}[i + k - 1 \bmod n, k]$;
  if $i = 1$
    then $C^{(k,1,j)}[k, j + k - 1 \bmod n] =$
      $C^{(k,1,j)}[1 + k - 1 \bmod n, j + k - 1 \bmod n]$
    else $C^{(k,i,j)}[k, j + k - 1 \bmod n] =$
      $C^{(k,i-1,j)}[k, j + k - 1 \bmod n]$;
  $C^{(k,i,j)}[i + k - 1 \bmod n, j + k - 1 \bmod n] =$
    $C^{(k,i,j)}[i + k - 1 \bmod n, j + k - 1 \bmod n] \bigoplus$
    $C^{(k,i,j)}[i + k - 1 \bmod n, k] \bigotimes C^{(k,i,j)}[k, j + k - 1 \bmod n]$
end_for /* Note: $i \bmod n \in \{1, 2, ..., n\}$ */

After labelling, there are also five data-dependence vectors: $d_1 = (0,0,1)^t$ for $(C^{(k,i,j)}[i+k-1 \bmod n, k], C^{(k,i,j-1)}[i+k-1 \bmod n, k])$, $d_2 = (0,1,0)^t$ for $(C^{(k,i,j)}[k, j+k-1 \bmod n], C^{(k,i-1,j)}[k, j+k-1 \bmod n])$, $d_3 = (1,-1,-1)^t$ for $(C^{(k,i,j)}[i+k-1 \bmod n, j+k-1 \bmod n], C^{(k',i',j')}[i'+k'-1 \bmod n, j'+k'-1 \bmod n])$, $d_4 = (1,-1,0)^t$ for $(C^{(k,i,n)}[i+k-1 \bmod n, n+k-1 \bmod n], C^{(k',i',n)}[i'+k'-1 \bmod n, k'])$ and $d_5 = (1,0,-1)^t$ for $(C^{(k,n,j)}[n+k-1 \bmod n, j+k-1 \bmod n], C^{(k',n,j')}[k', j'+k'-1 \bmod n])$.

We have d_3, because $C[i+k-1 \bmod n, j+k-1 \bmod n] = C[i'+k'-1 \bmod n, j'+k'-1 \bmod n]$, $i+k-1 \bmod n = i'+k'-1 \bmod n$, $j+k-1 \bmod n = j'+k'-1 \bmod n$, and $k' = k-1$, thus $(k,i,j)^t - (k',i',j')^t = (1,-1,-1)^t$. Similarly we have d_4, because $C[i+k-1 \bmod n, n+k-1 \bmod n] = C[i'+k'-1 \bmod n, k']$, $i+k-1 \bmod n = i'+k'-1 \bmod n$, $n+k-1 \bmod n = k'$, and $k' = k-1$, thus $(k,i,j)^t-(k',i',j')^t = (1,-1,0)^t$. Finally, we have d_5, because $C[n+k-1 \bmod n, j+k-1 \bmod n] = C[k', j'+k'-1 \bmod n]$, $n+k-1 \bmod n = k'$, $j+k-1 \bmod n = j'+k'-1 \bmod n$, and $k' = k-1$, thus $(k,i,j)^t - (k',i',j')^t = (1,0,-1)^t$. The data-dependence graph for $n = 3$ is shown in Fig-12. Unlike in the Warshall-Floyd algorithm, all the data-dependence vectors are constant.

Following our method, we obtain $\mathbf{H}_6^{\mathbf{L}} = (3,1,1)$, and if we let $\mathbf{S}_6 = (0,1,1)$, $\mathbf{H}_6^1 = (a,b,c)$, then from Condition 1, we have $a,b,c > 0$, and then from Condition 3', we have $\frac{c}{1} = c_1$, $\frac{b}{1} = c_2$, $\frac{a-b-c}{-2} = c_3$, $\frac{a-b}{-1} = c_4$ and $\frac{a-c}{-1} = c_5$, where all of the c_1 to c_5 are integers, after that, from Condition 5', we have $c_i \neq c_j$ for all $i \neq j$ and $1 \leq i,j \leq 5$. Then, modifying $\mathbf{H}_6^{\mathbf{L}} = (3,1,1)$, we let $b = 2$ and $c = 1$. After that, from Conditions 2 and 5, $\frac{\mathbf{H}_6^1 (x,y,z)^t}{\mathbf{S}_6 (x,y,z)^t} \neq 1, 2, \frac{a-3}{-2}, \frac{a-2}{-1}, \frac{a-1}{-1}$, for $(x,y,z)^t = I_2 - I_1$ and $I_2 - I_1 \neq md_i$. Then we obtain the following five inequalities:

$$ax + y \neq 0 \quad (\text{unless } (x,y,z)^t = md_1 \text{ for some } m) \tag{10}$$

$$ax - z \neq 0 \quad (\text{unless } (x,y,z)^t = md_2 \text{ for some } m) \tag{11}$$

$$a(2x+y+z) \neq (z-y) \quad (\text{unless } (x,y,z)^t = md_3 \text{ for some } m) \tag{12}$$

$$-a(x+y) \neq (a-1)z \quad (\text{unless } (x,y,z)^t = md_4 \text{ for some } m) \tag{13}$$

$$-a(x+z) \neq (a+1)y \quad (\text{unless } (x,y,z)^t = md_5 \text{ for some } m) \tag{14}$$

Finally, if $a \geq n$, $a > 2(n-1)$ and a is odd, we can get $(a,2,1)$ as a solution for $\mathbf{H}_6^1$.

As an example, for $n = 3$, let $a = 5$. Then the data-flow diagram of the linear-array algorithm $\mathbf{H}_6^1 = (5,2,1)$ and $\mathbf{S}_6 = (0,1,1)$ is shown in Fig-13. This mapping maps $(k,i,j)^t$ to step $(5k+2i+j)$ and executes it in PE_{i+j}. The following describes how the linear array works:

1. Each PE has five data links, data links 1 and 2 flow from left to right, data links 3, 4, and 5 flow from right to left. In addition, there are no shifting registers for data links 1 and 3,

one shifting register for data link 2, two shifting registers for data link 4, and three shifting registers for data link 5.

2. Only one input stream C is fed into the array. It satisfies Corollary 5 for the entrance time and is fed into data link 3 from right to left.

3. In each index execution we need three data tokens: z_3 ($C[i+k-1 \bmod n, j+k-1 \bmod n]$), z_1 ($C[i+k-1 \bmod n, k]$), z_2 ($C[k, j+k-1 \bmod n]$) and generate a new result r ($C[i+k-1 \bmod n, j+k-1 \bmod n]$). We will use $x \leftarrow l_i$ to represent that the PE gets x from the data link i and will use $y \rightarrow l_j$ to represent that the PE puts y onto the data link j. Then, when $k = 1$, we have the following functions in each cell when performing the index $(i, j, k)^t$:

	$j = 1$		$1 < j < n$		$j = n$	
$i = 1$	$z_1 \leftarrow z_3$	$r \rightarrow l_1$	$z_1 \leftarrow l_1$	$l_1 \rightarrow l_1$	$z_1 \leftarrow l_1$	$l_1 \rightarrow l_4$
	$z_2 \leftarrow z_3$	$r \rightarrow l_2$	$z_2 \leftarrow z_3$	$r \rightarrow l_2$	$z_2 \leftarrow z_3$	$r \rightarrow l_2$
	$z_3 \leftarrow l_3$	$r \rightarrow l_3$	$z_3 \leftarrow l_3$	$r \rightarrow l_3$	$z_3 \leftarrow l_3$	$r \rightarrow l_3$
$1 < i < n$	$z_1 \leftarrow z_3$	$r \rightarrow l_1$	$z_1 \leftarrow l_1$	$l_1 \rightarrow l_1$	$z_1 \leftarrow l_1$	$l_1 \rightarrow l_4$
	$z_2 \leftarrow l_2$	$l_2 \leftarrow l_2$	$z_2 \leftarrow l_2$	$l_2 \rightarrow l_2$	$z_2 \leftarrow l_2$	$l_2 \rightarrow l_2$
	$z_3 \leftarrow l_3$	$r \rightarrow l_3$	$z_3 \leftarrow l_3$	$r \rightarrow l_3$	$z_3 \leftarrow l_3$	$r \rightarrow l_3$
$i = n$	$z_1 \leftarrow z_3$	$r \rightarrow l_1$	$z_1 \leftarrow l_1$	$l_1 \rightarrow l_1$	$z_1 \leftarrow l_1$	$l_1 \rightarrow l_4$
	$z_2 \leftarrow l_2$	$l_2 \leftarrow l_5$	$z_2 \leftarrow l_2$	$l_2 \rightarrow l_5$	$z_2 \leftarrow l_2$	$l_2 \rightarrow l_5$
	$z_3 \leftarrow l_3$	$r \rightarrow l_3$	$z_3 \leftarrow l_3$	$r \rightarrow l_3$	$z_3 \leftarrow l_3$	$r \rightarrow l_3$

When $2 \leq k \leq n$, we have the following functions in each cell when performing the index $(i, j, k)^t$:

	$j = 1$		$1 < j < n$		$j = n$	
$i = 1$	$z_1 \leftarrow z_3$	$r \rightarrow l_1$	$z_1 \leftarrow l_1$	$l_1 \rightarrow l_1$	$z_1 \leftarrow l_1$	$l_1 \rightarrow l_4$
	$z_2 \leftarrow z_3$	$r \rightarrow l_2$	$z_2 \leftarrow z_3$	$r \rightarrow l_2$	$z_2 \leftarrow z_3$	$r \rightarrow l_2$
	$z_3 \leftarrow l_3$	$r \rightarrow l_3$	$z_3 \leftarrow l_3$	$r \rightarrow l_3$	$z_3 \leftarrow l_4$	$r \rightarrow l_3$
$1 < i < n$	$z_1 \leftarrow z_3$	$r \rightarrow l_1$	$z_1 \leftarrow l_1$	$l_1 \rightarrow l_1$	$z_1 \leftarrow l_1$	$l_1 \rightarrow l_4$
	$z_2 \leftarrow l_2$	$l_2 \rightarrow l_2$	$z_2 \leftarrow l_2$	$l_2 \rightarrow l_2$	$z_2 \leftarrow l_2$	$l_2 \rightarrow l_2$
	$z_3 \leftarrow l_3$	$r \rightarrow l_3$	$z_3 \leftarrow l_3$	$r \rightarrow l_3$	$z_3 \leftarrow l_4$	$r \rightarrow l_3$
$i = n$	$z_1 \leftarrow z_3$	$r \rightarrow l_1$	$z_1 \leftarrow l_1$	$l_1 \rightarrow l_1$	$z_1 \leftarrow l_1$	$l_1 \rightarrow l_4$
	$z_2 \leftarrow l_2$	$l_2 \rightarrow l_5$	$z_2 \leftarrow l_2$	$l_2 \rightarrow l_5$	$z_2 \leftarrow l_2$	$l_2 \rightarrow l_5$
	$z_3 \leftarrow l_5$	$r \rightarrow l_3$	$z_3 \leftarrow l_5$	$r \rightarrow l_3$	$z_3 \leftarrow 1$	$r \rightarrow l_3$

4. The output results also appear in data link 3. □

The time complexity of this algorithm is $O(n^2)$, the number of the PEs used is $O(n)$. In each PE, we need $O(n)$ shifting registers for data links 3, 4 and 5, requiring the total storage of $O(n^2)$. Since, in this linear-array algorithm the number of the local registers in each PE is a function of the problem size n, this algorithm is not modularly extensible.

We have also found that $\mathbf{H}_7^1 = (a', 3, 1)$ and $\mathbf{S_7} = (3, 1, 1)$ can be easily partitioned, where $a' > 24(n-1)$, $a' > 2n + 9$, and a' is odd. However, $(\mathbf{H}_7^1, \mathbf{S_7})$ is too not modularly extensible. We ommit the details from this presentation.

8 Conclusions

A systematic method for synthesizing linear-array implementations from nested loop algorithms has been presented in this paper. The method was based on a set of formal necessary and sufficient conditions on the sequential algorithm and feasible transformations that were listed in the paper. This method can be used down to register level.

An important contribution of this paper is the classification of data-dependence vectors based on the "Zero-One-Infinite" property. Having this classification, we can understand the characters of the tokens in data streams. It also allows us to formulate conditions on the target linear array. That is, we can determine whether tokens can be destroyed or not, whether tokens can be pipelined or not, and whether a PE needs additional I/O ports or not.

We also find tight bounds on time complexity of linear array implementations for an important class of algorithms. Specifically, time complexity = Θ(storage complexity). Such algorithms include matrix multiplication, L-U decomposition, inversion of nonsingular triangular matrix, matrix orthogonal triangularization, a version of transitive closure algorithm [7], a version of Discrete Fourier Transform (DFT) [24], bubble sort, and others. We also provide a technique to determine whether linear array implementations of these algorithms have both the optimal time complexity and optimal storage complexity.

We studied a partition model for designing algorithms that can be partitioned. Our method for partitioning is quite general, although we need to impose the condition that all data streams flow in the same direction.

Synthesis of families of linear array implementation for matrix multiplication algorithms was used to illustrate our method. We were thus able to show the interplay between the formal conditions listed and various properties of the linear array implementations. In addition we used the method to synthesize a linear array algorithm for a variant of the Warshall-Floyd algorithm [19].

As in this paper we restricted ourselves to designing linear arrays with constant I/O, we did not consider the case where $\mathbf{S}d_i = 0$. Our methodology can, however, be extended to handle this

case too. After doing that, we can synthesize all 1-D time hyperplane and one-wavefront 1-D space hyperplane mappings to systolic implementations. That is, if the data streams of systolic linear array implementations "do not change" directions during the execution, then we also can synthesize those implementations.

Our method can be used in two ways. It can transform a p nested loop algorithm onto a new linear array implementation. It also can synthesize a linear array implementation by listing basic constraints on a PE first. Thus our method contributes towards the utilization of programmable linear arrays. That is, in the model of general $\mathbf{S}d_i$, we can design only one type of a PE with appropriate number of data links and local registers, which is subsequently used to synthesize a single linear array that can solve matrix multiplication, L-U decomposition, triangular linear system, triangular matrix inversion, DFT, convolution, longest common subsequence, transitive closure, matrix-vector multiplication, bubble sort, and others.

As our method is able to synthesize large families of feasible linear array implementations, a software tool analogous to that of Moldovan's [29] could be used to help in selecting implementations optimizing additional criteria.

Finally, the problem of transforming the algorithms with the non-constant data dependencies to linear array algorithms is still not satisfactorily solved, even though it may be possible to handle them on an *ad hoc* basis as we did for the Warshall-Floyd algorithm.

9 Appendix : Summary of the Method

In this appendix, we summarize our mapping method as a procedure with seven steps. In what follows, the algorithm model is $Ag = (I^p, V_{Ag}, F_{Ag})$. We use symbol d_i to denote the i-th data-dependence vector; $\bar{I}$, I_1, and I_2, the indices in the index set $I^p = \{(i_1, i_2, \ldots, i_p)^t | l_j \leq i_j \leq u_j \ for\ j = 1,2,\ldots,p\}$; $\mathbf{H^1} = \mathbf{H^L} + \Pi$, where $\mathbf{H^L}$ is Lamport's hyperplane, Π is the assistant hyperplane related to $\mathbf{H^L}$, and $\mathbf{H^1}$ is the hyperplane for 1-D linear array; $\mathbf{S}$, the space hyperplane. The linear-array model is $Ar = (M, K, T_{Ar}, B_{Ar}, F_{Ar})$; t_i, b_i denotes the direction and the number of the shifting registers in each PE of the data link i, respectively.

Procedure
Input : A p nested loop algorithm $Ag = (I^p, V_{Ag}, F_{Ag})$.
Output: A linear-array algorithm $(\mathbf{H^1}, \mathbf{S})$. Where,
$\mathbf{H^1}$ is a 1-D time hyperplane, and $\mathbf{S}$ is a 1-D space hyperplane.
Input or Output: A linear array $Ar = (M, K, T_{Ar}, B_{Ar}, F_{Ar})$ can be
either as an input for restricted interconnection primitives
or as an output of a new linear array.

1. Index all variables in the algorithm, that is, eliminate broadcasting of data and add the missing index to each variable.

2. Find the set of data-dependence vectors. Each data-dependence vector d_i will correspond to data link i of the linear array.

3. Compute $\mathbf{H^1}$ and $\mathbf{S}$.
   We use a heuristic method:

   First, find Lamport's hyperplane $\mathbf{H^L} = (\alpha_1, \alpha_2, \ldots, \alpha_p)$, such that $\mathbf{H^L} d_i > 0$ for all data-dependence vectors, and it satisfies $\mathbf{H^L} = \min_{\bar{\mathbf{H}}}\{\max\{|\bar{\mathbf{H}}(I_2 - I_1)| \mid I_1, I_2 \in I^p\}\}$.

   Second, let the space mapping $\mathbf{S} = (s_1, s_2, \ldots, s_p)$, where $s_1, s_2, \ldots$, and s_p are variables with values to be determined later.

   Third, let the assistant hyperplane $\mathbf{\Pi} = (\pi_1, \pi_2, \ldots, \pi_p)$, where $\pi_1, \pi_2, \ldots$, and π_p are also variables with values to be determined later.

   Fourth, let $\mathbf{H^1} = \mathbf{H^L} + \mathbf{\Pi} = (\alpha_1 + \pi_1, \alpha_2 + \pi_2, \ldots, \alpha_p + \pi_p)$.
   (Note: We found that it is easy to derive $\mathbf{H^1}$ by modifying Lamport's $\mathbf{H^L}$, other methods for finding $\mathbf{H^1}$ independently are possible.)

   Fifth, define the relations between $\mathbf{H^1}$ and $\mathbf{S}$. In order to preserve the data-dependence relations and to relate them to the linear-array model, $\mathbf{H^1}$ and $\mathbf{S}$ must satisfy the following constraints:

   (a) $\mathbf{H^1} d_i > 0$ for every data-dependence vector d_i.

   (b) If I_1 and I_2 are two indices of I^p, then
   $\begin{pmatrix} \mathbf{H^1} \\ \mathbf{S} \end{pmatrix}(I_2 - I_1) \neq \begin{pmatrix} 0 \\ 0 \end{pmatrix}$; that is, $\begin{pmatrix} \mathbf{H^1} \\ \mathbf{S} \end{pmatrix}(x_1, \ldots, x_p)^t \neq \begin{pmatrix} 0 \\ 0 \end{pmatrix}$
   for $(p_j - q_j) \leq x_j \leq (q_j - p_j)$ and $j = 1, \ldots, p$.

   (c) if $d_i \neq m d_j$ and $d_j \neq m d_i$ for any m, then $\frac{\mathbf{H^1} d_i}{\mathbf{S} d_i} \neq \frac{\mathbf{H^1} d_j}{\mathbf{S} d_j}$.

   (d) If $(I_2 - I_1) \neq m d_i$ for any m, then $\mathbf{H^1}(I_2 - I_1)\mathbf{S} d_i \neq \mathbf{S}(I_2 - I_1)\mathbf{H^1} d_i$.

   (e) $\frac{\mathbf{H^1} d_i}{\mathbf{S} d_i}$ must be an integer for every data-dependence vector d_i.

   (f) If there are b_i shifting registers in each PE for the data link i, then $b_i = |\frac{\mathbf{H^1} d_i}{\mathbf{S} d_i}| - 1$.

   (g) If $\mathbf{S} d_i > 0$ then $t_i = 1$ and the data stream i will be fed into the linear array at $PE_{\min\{\mathbf{S}I_1 | I_1 \in I^p\}}$. If $\mathbf{S} d_i < 0$ then $t_i = -1$ and the data stream i will be fed into the linear array at $PE_{\max\{\mathbf{S}I_2 | I_2 \in I^p\}}$. (Note: If $\mathbf{S} d_i = 0$ then $t_i = 0$ and the data stream will

be fixed in the PEs. However, in this paper, we chose not to consider this case, although it is a simple extension.)

Note: (a)-(d) guarantee no data conflicts in the data flow; (e)-(g) relate the linear-array algorithm $(\mathbf{H^1}, \mathbf{S})$ to the linear-array model Ar.

4. Determine the trade-offs between the time, the number of PEs, and the number of the local shifting registers in each PE.

   (a) Given a space mapping $\mathbf{S} = (s_1, s_2, \ldots, s_p)$, $\mathbf{H^1}$ must satisfy all of the constraints in step 3.

   (b) Given the number of the shifting registers of the PE, from constraints (e)-(g) in step 3. $\mathbf{H^1}$ and $\mathbf{S}$ must satisfy, $\frac{\mathbf{H^1}d_i}{\mathbf{S}d_i} = t_i(b_i + 1)$.

   (c) If we want to obtain an algorithm with minimum execution time, then we add the constraint $\mathbf{H^1} = \min_{\bar{\mathbf{H}}^1}\{\max\{|\bar{\mathbf{H}}^1(I_2 - I_1)| \mid I_1, I_2 \in I^p\}\}$.

   (d) If we want to obtain an algorithm with minimum number of PEs, then we add the constraint $\mathbf{S} = \min_{\bar{\mathbf{S}}}\{\max\{|\bar{\mathbf{S}}(I_2 - I_1)| \mid I_1, I_2 \in I^p\}\}$.

   Note that, of course, it may not in general be possible to satisfy the two objectives (c) and (d) simultaneously.

5. Map the algorithm into the linear array:

   - The number of PEs is $M = \max\{|\mathbf{S}(I_2 - I_1)| \mid I_1, I_2 \in I^p\} + 1$.
     Note: Although the PEs are numbered here from $\min\{\mathbf{S}I_1 | I_1 \in I^p\}$ to $\max\{\mathbf{S}I_2 | I_2 \in I^p\}$, we can renumber them from 1 to M.
   - The execution time is $T = O(\max\{|\mathbf{H^1}(I_2 - I_1)| \mid I_1, I_2 \in I^p\} + M(1 + \sum_{i=1}^{K} b_i))$.
   - Define the entrance time of each data token:
     Suppose a variable x, whose data-dependence vector is d_i of type 1, is used in $\bar{I}$ and enters $PE_{\mathbf{S}\bar{I}}$ at time $\mathbf{H^1}\bar{I}$.
     If $\mathbf{S}d_i > 0$, then x is fed into the linear array at $PE_{\min\{\mathbf{S}I_1 | I_1 \in I^p\}}$ at time

$$\mathbf{H^1}\bar{I} - (\mathbf{S}\bar{I} - \min\{\mathbf{S}I_1 | I_1 \in I^p\})\frac{\mathbf{H^1}d_i}{\mathbf{S}d_i}.$$

     If $\mathbf{S}d_i < 0$, then x is fed into $PE_{\max\{\mathbf{S}I_2 | I_2 \in I^p\}}$ at time

$$\mathbf{H^1}\bar{I} - (\max\{\mathbf{S}I_2 | I_2 \in I^p\} - \mathbf{S}\bar{I})\frac{\mathbf{H^1}d_i}{\mathbf{S}d_i}.$$

- We can execute the function $F_{Ar} = F_{Ag}$ in $PE_{\mathbf{S}\bar{I}}$ at time $\mathbf{H^1}\bar{I}$.

6. Consider partitioning of the algorithm, when the size of the array required for the unpartitioned algorithm is larger than that of the available linear array:

   If all of the data streams have the same direction, i.e. for all d_i, $\mathbf{S}d_i > 0$, and q is the number of PEs in the available linear array, then we have a partitioned algorithm $(\mathbf{H_q^1}, \mathbf{S_q})$ as follow:

   - $\mathbf{H_q^1}\bar{I} = \mathbf{H^1}\bar{I}$ in phase $\lceil(\mathbf{S}\bar{I} - \min\{\mathbf{S}I_1 | I_1 \in I^p\} + 1)/q\rceil$, and
   - $\mathbf{S_q}\bar{I} = (\mathbf{S}\bar{I} - \min\{\mathbf{S}I_1 | I_1 \in I^p\} + 1) \bmod q$. ($a \bmod b \in \{1, 2, \ldots, b\}$.)

7. Analyze the performance.

References

[1] A.V.Aho, J.E.Hopcraft and J.D.Ullman, *"The Design and Analysis of Computer Algorithms"*, Reading, MA: Addison-Wesley, 1974.

[2] M.A. Annaratone, F. Bitz, E. Clune, H.T. Kung, P. Maulik, O. Menzilcioglu H. Rabas, and J.A. Webb, *"Applications and Algorithm Partitioning on Warp"*, *IEEE 1987 COMPCON*, Feb. 23-27, pp. 272-275.

[3] M.A. Annaratone, E. Arnould, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu and J.A. Webb, *"The WARP Computer: Architecture, Implementation, and Performance"*, *IEEE Trans. on Computers, Vol. C-36, No. 12*, Dec. 1987, pp. 1523-1538.

[4] U.Banerjee, S.C.Chen, D.J.Kuck, R.A.Towle, *"Time and Parallel Processor Bounds for FORTRAN Like Loops"*, *IEEE Trans. on Computers, Vol. C-28*, Sep. 1979, pp. 660-670.

[5] R.P. Brent and H.T. Kung, *"The Area-Time Complexity of Binary Multiplication"*, *JACM*, 1981, pp. 521-534.

[6] M.C. Chen, *"The Generation of a Class of Multipliers: Synthesizing Highly Parallel Algorithms in VLSI"*, *IEEE Trans. on Computers, Vol. C-37, No. 3*, March 1988, pp. 329-338.

[7] L.J. Guibas, H.T. Kung and C.D Thompson, *"Direct VLSI Implementation of Combinatorial Algorithms"*, *CALTECH Conf. on VLSI*, Jan. 1979, pp. 509-525.

[8] B.Hochet, P.Quinton and Y.Robert, *"Systolic Solution of Linear Systems over GF(p) with Partial Pivoting"*, *IEEE CH2419-0*, 1987, pp. 161-168.

[9] K.Hwang and Y.H.Cheng, *"Partitioned Matrix Algorithms for VLSI Arithmetic Systems"*, *IEEE Trans. on Computers, Vol. C-31*, Dec. 1982, pp. 1215-1224.

[10] K.Hwang and F.A.Briggs, *"Computer Architecture and Parallel Processing"*, Mcgraw-Hill. New York, 1984.

[11] Z.M.Kedem, *"Optimal Allocation of Area for Single-Chip Computations"*, *SIAM J. Comput., Vol. 14*, Aug. 1985, pp. 730-743.

[12] R. H. Kuhn, *"Transforming Algorithms for Single-Stage and VLSI Architectures"*, *Proceedings of the workshop on Interconnection Networks for Parallel and Distributed Processing, IEEE CH1560-2*, 1980, pp. 11-19.

[13] A.V.Kulkarni and D.W.L.Yen, *"Systolic Processing and an Implementation for Signal and Image Processing"*, *IEEE Trans. on Computers. Vol. C-31*, Oct. 1982, pp. 1000-1009.

[14] H.T.Kung, C.E.Leiserson, *"Systolic Arrays for (VLSI)"*, *Technical Report CMU-CS-79-103*, CMU, April 1978.

[15] H.T.Kung, *"Let's Design Algorithms for VLSI Systems"*, *Technical Report CMU-CS-79-151*, CMU, January 1979.

[16] H.T.Kung, *"Why Systolic Architectures ?"*, *IEEE Computer*, Jan. 1982, pp.37-46.

[17] H.T.Kung, *"Systolic algorithms for the CMU WARP Processor"*, *Proc. Seventh Int. Conf. Pattern Recognition*, July 1984, pp. 570-577.

[18] S.Y.Kung, *"On Supercomputing with Systolic/Wavefront Array Processors"*, *Proc. IEEE, Vol. 72*, July 1984.

[19] S.Y.Kung, S.C.Lo and P.S.Lewis, *"Optimal Systolic Design for the Transitive Closure and the Shortest Path Problems"*, *IEEE Trans. on Computers, Vol. C-36* May 1987, pp. 603-614.

[20] L.Lamport, *"The Parallel Execution of Do Loops"*, *CACM*, Feb. 1974, pp. 83-93.

[21] PeiZong Lee, *"A New VLSI Synthesis Method"*, *M.S. Thesis*, National Tsing-Hua University, Taiwan, 1984.

[22] G. Li and B.W. Wah, *"The Design of Optimal Systolic Arrays"*, *IEEE Trans. on Computers, Vol. C-34, No. 1*, Jan. 1985, pp. 66-77.

[23] Y.J.Ma, J.F.Wang and J.Y.Lee, *"Systolic Array Mapping of Sequential Algorithm for VLSI Architecture"*, *Proceedings of ICS*, Tainan, Taiwan, R.O.C., Dec. 1986, pp. 865-874.

[24] C. Mead and L. Conway, *"Introduction to VLSI Systems"*, Addison-Wesley, 1980.

[25] W.L. Miranker and A. Winkler, *"Spacetime representations of computational structures"*, *Computing 32*, 1984, pp. 93-114.

[26] D.I.Moldovan, *"On the Analysis of VLSI Systems"*, *IEEE Trans. on Computers, Vol. C-31*, Nov. 1982, pp. 1121-1126.

[27] D.I.Moldovan, *"On the design of algorithms for VLSI systolic arrays"*, *Proc. IEEE Vol. 71*, Jan. 1983, pp. 113-120.

[28] D.I.Moldovan and J.A.Fortes, *"Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays"*, *IEEE Trans. on Computers, Vol. C-35*, January 1986, pp. 1-12.

[29] D.I.Moldovan, *"ADVIS: A Software Package for the Design of Systolic Arrays"*, *IEEE Trans. on Computer-Aided Design, Vol. CAD-6*, Jan. 1987, pp. 33-40.

[30] P. Quinton, *"Automatic synthesis of systolic arrays from uniform recurrent equations"*, *Proc. 11th Annu. Symp. Comput. Architecture*, 1984, pp. 208-214.

[31] I.Ramakrishnan. D.Fussell and A.Silberschatz. *"A Linear Array Matrix Multiplication Algorithm"*, *Proc. 20th annu. Allerton Conf. on Comput., Control and Commun.*, Oct. 1982.

[32] I.Ramakrishnan and P.Varman, *"Modular Matrix Multiplication on a Linear Array"*, *IEEE Trans. on Computers, Vol. C-33*, Nov. 1984, pp. 952-958.

[33] I.Ramakrishnan, D.Fussell and A.Silberschatz, *"Mapping Homogeneous Graphs on Linear Arrays"*, *IEEE Trans. on Computers, Vol. C-35*, March 1986, pp. 198-209.

[34] Gunter Rote, *"A Systolic Array Algorithm for the Algebraic Path Problem"*, *Computing 34*, 1985, pp. 191-219.

[35] C.D. Thompson, *"Area-Time Complexity for VLSI"*, *Proc. 11-th ACM STOC*, April 1979, pp. 81-88.

[36] J.D.Ullman, *"Computational Aspects of VLSI"*, Computer Science Press Inc., 1984.

[37] P.Varman and I.Ramakrishnan, *"Dynamic Programming and Transitive Closure on Linear Pipelines"*, *Proc. of the 1984 International Conf. on Parallel Processing*, pp. 359-364.

[38] P.Varman and I.Ramakrishnan, *"On Matrix Multiplication Using Array Processors"*, *Automata, Languages and Programming, 12th Colloquium*, Nafplion, Greece, July 1985, pp. 487-496.

[39] P.Varman and I.Ramakrishnan, *"A Fault-Tolerant VLSI Matrix Multiplier", TR 85/29,* State Univ. of NY at Stony Brook, Oct. 1985.

[40] P.Varman and I.Ramakrishnan. *"Synthesis of an Optimal Family of Matrix Multiplication Algorithms on Linear Arrays", IEEE Trans. on Computers, Vol. C-35,* Nov. 1986, pp. 989-996.

[41] Y. Wong and J. Delosme, *"Optimal Systolic Implementations of N-Dimensional Recurrences", IEEE 1985 ICCD,* pp. 618-621.

[42] C.B.Yang and R. C. T. Lee, *"The Mapping of 2-D Array Processors to 1-D Array Processors", International Symposium on VLSI Technology, Systems and Applications,* Taipei, Taiwan, R.O.C., 1985. pp. 216-220.

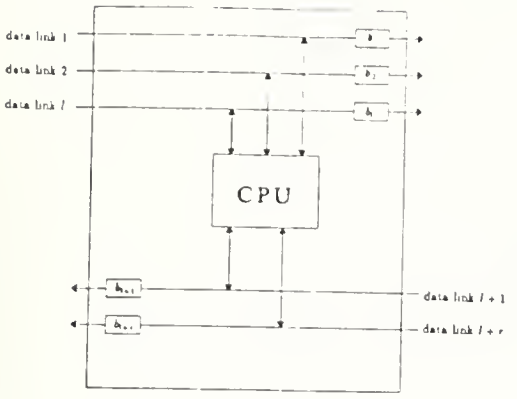


Fig. 1. Processing Element (PE)

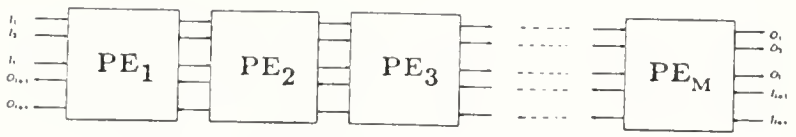


Fig. 2. Linear array

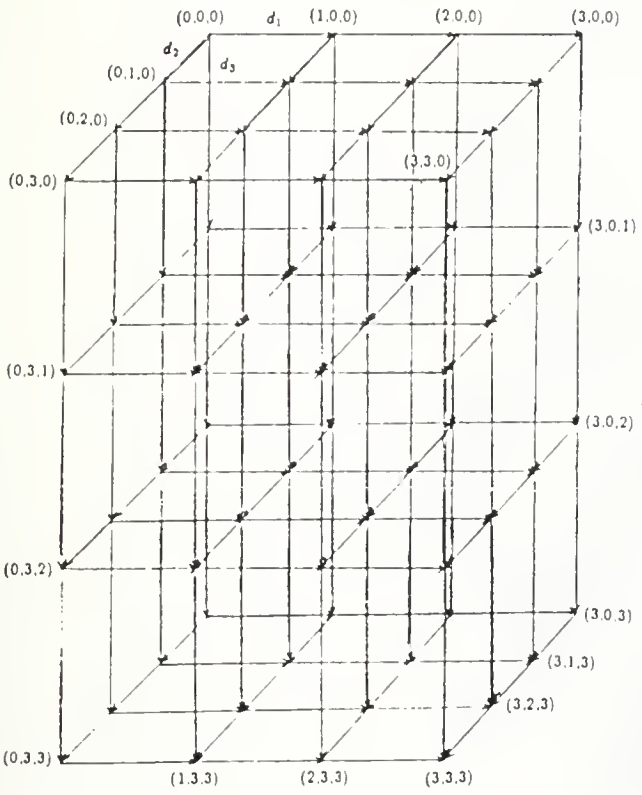


Fig. 3 data dependence graph

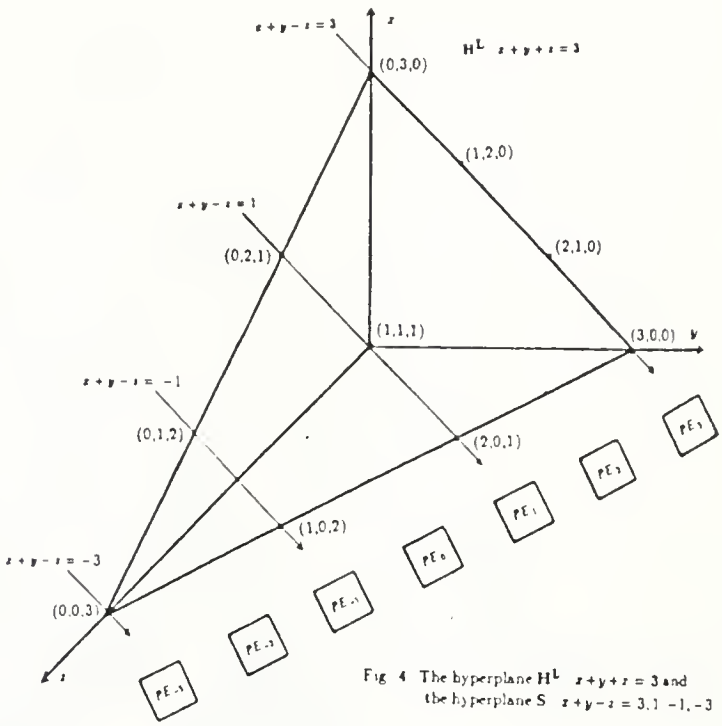


Fig. 4 The hyperplane H^L $x+y+z=3$ and the hyperplane S $x+y-z=3,1,-1,-3$

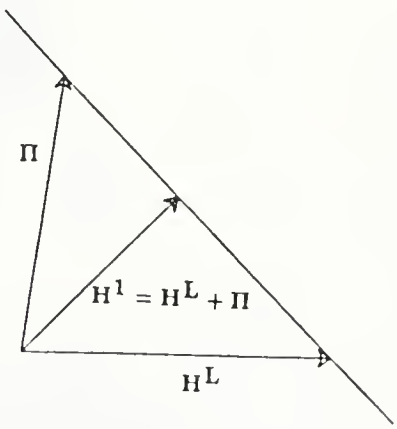
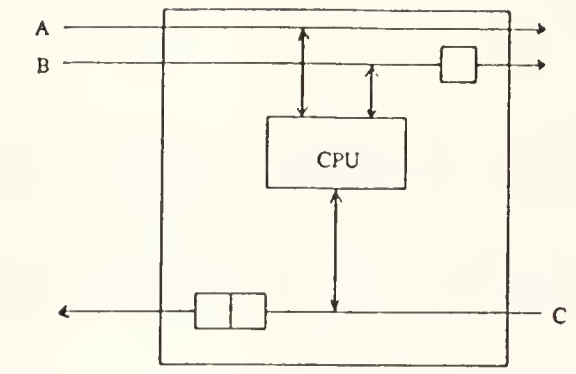
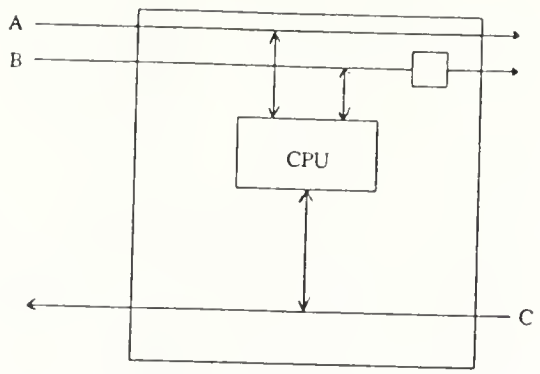


Fig. 5. The normal direction of H^L, Π, and H^1



$2i+j+3k$	PE_{-3}	PE_{-2}	PE_{-1}	PE_0	PE_1	PE_2	PE_3	PE_4	PE_5	PE_6
0				$0\ A_{00}$ $0\ B_{00}$ $0\ C_{00}$						
1					$0\ A_{00}$ $1\ B_{01}$ $0\ C_{01}$					
2					$1\ A_{10}$ $0\ B_{00}$ $0\ C_{10}$	$0\ A_{00}$ $2\ B_{02}$ $0\ C_{02}$				
3			$0\ A_{01}$ $0\ B_{10}$ $1\ C_{00}$			$1\ A_{10}$ $1\ B_{01}$ $0\ C_{11}$	$0\ A_{00}$ $3\ B_{03}$ $0\ C_{03}$			
4				$0\ A_{01}$ $1\ B_{11}$ $1\ C_{01}$		$2\ A_{20}$ $0\ B_{00}$ $0\ C_{20}$	$1\ A_{10}$ $2\ B_{02}$ $0\ C_{12}$			
5				$1\ A_{11}$ $0\ B_{10}$ $1\ C_{10}$	$0\ A_{01}$ $2\ B_{12}$ $1\ C_{02}$		$2\ A_{20}$ $1\ B_{01}$ $0\ C_{21}$	$1\ A_{10}$ $3\ B_{03}$ $0\ C_{13}$		
6		$0\ A_{02}$ $0\ B_{20}$ $2\ C_{00}$			$1\ A_{11}$ $1\ B_{11}$ $1\ C_{11}$	$0\ A_{01}$ $3\ B_{13}$ $1\ C_{03}$	$3\ A_{30}$ $0\ B_{00}$ $0\ C_{30}$	$2\ A_{20}$ $2\ B_{02}$ $0\ C_{23}$		
7			$0\ A_{02}$ $1\ B_{21}$ $2\ C_{01}$		$2\ A_{21}$ $0\ B_{10}$ $1\ C_{20}$	$1\ A_{11}$ $2\ B_{12}$ $1\ C_{13}$		$3\ A_{30}$ $1\ B_{01}$ $0\ C_{31}$	$2\ A_{20}$ $3\ B_{03}$ $0\ C_{33}$	
8			$1\ A_{12}$ $0\ B_{20}$ $2\ C_{10}$	$0\ A_{02}$ $2\ B_{22}$ $2\ C_{02}$		$2\ A_{21}$ $1\ B_{11}$ $1\ C_{21}$	$1\ A_{11}$ $3\ B_{13}$ $1\ C_{13}$		$3\ A_{30}$ $2\ B_{02}$ $0\ C_{32}$	
9	$0\ A_{03}$ $0\ B_{30}$ $3\ C_{00}$			$1\ A_{12}$ $1\ B_{21}$ $2\ C_{11}$	$0\ A_{02}$ $3\ B_{23}$ $2\ C_{03}$	$3\ A_{31}$ $0\ B_{10}$ $1\ C_{30}$	$2\ A_{21}$ $2\ B_{12}$ $1\ C_{23}$			$3\ A_{30}$ $3\ B_{03}$ $0\ C_{33}$
10		$0\ A_{03}$ $1\ B_{31}$ $3\ C_{01}$		$2\ A_{22}$ $0\ B_{20}$ $2\ C_{20}$	$1\ A_{12}$ $2\ B_{22}$ $2\ C_{12}$		$3\ A_{31}$ $1\ B_{11}$ $1\ C_{31}$	$2\ A_{21}$ $3\ B_{13}$ $1\ C_{33}$		
11		$1\ A_{13}$ $0\ B_{30}$ $3\ C_{10}$	$0\ A_{03}$ $2\ B_{32}$ $3\ C_{02}$		$2\ A_{22}$ $1\ B_{21}$ $2\ C_{31}$	$1\ A_{12}$ $3\ B_{23}$ $2\ C_{13}$		$3\ A_{31}$ $2\ B_{12}$ $1\ C_{32}$		
12			$1\ A_{13}$ $1\ B_{31}$ $3\ C_{11}$	$0\ A_{03}$ $3\ B_{33}$ $3\ C_{03}$	$3\ A_{32}$ $0\ B_{20}$ $2\ C_{30}$	$2\ A_{22}$ $2\ B_{22}$ $2\ C_{23}$			$3\ A_{31}$ $3\ B_{13}$ $1\ C_{33}$	
13			$2\ A_{23}$ $0\ B_{30}$ $3\ C_{20}$	$1\ A_{13}$ $2\ B_{32}$ $3\ C_{13}$		$3\ A_{32}$ $1\ B_{21}$ $2\ C_{31}$	$2\ A_{22}$ $3\ B_{23}$ $2\ C_{23}$			
14				$2\ A_{23}$ $1\ B_{31}$ $3\ C_{21}$	$1\ A_{13}$ $3\ B_{33}$ $3\ C_{13}$		$3\ A_{32}$ $2\ B_{22}$ $2\ C_{32}$			
15				$3\ A_{33}$ $0\ B_{30}$ $3\ C_{30}$	$2\ A_{23}$ $2\ B_{32}$ $3\ C_{22}$			$3\ A_{32}$ $3\ B_{23}$ $2\ C_{33}$		
16					$3\ A_{33}$ $1\ B_{31}$ $3\ C_{31}$	$2\ A_{23}$ $3\ B_{23}$ $3\ C_{23}$				
17						$3\ A_{33}$ $2\ B_{32}$ $3\ C_{32}$				
18							$3\ A_{33}$ $3\ B_{33}$ $3\ C_{33}$			

Fig. 6. The data-flow diagram of $\Pi^1 = (2, 1, 3)$ and $S = (1, 1, -1)$.



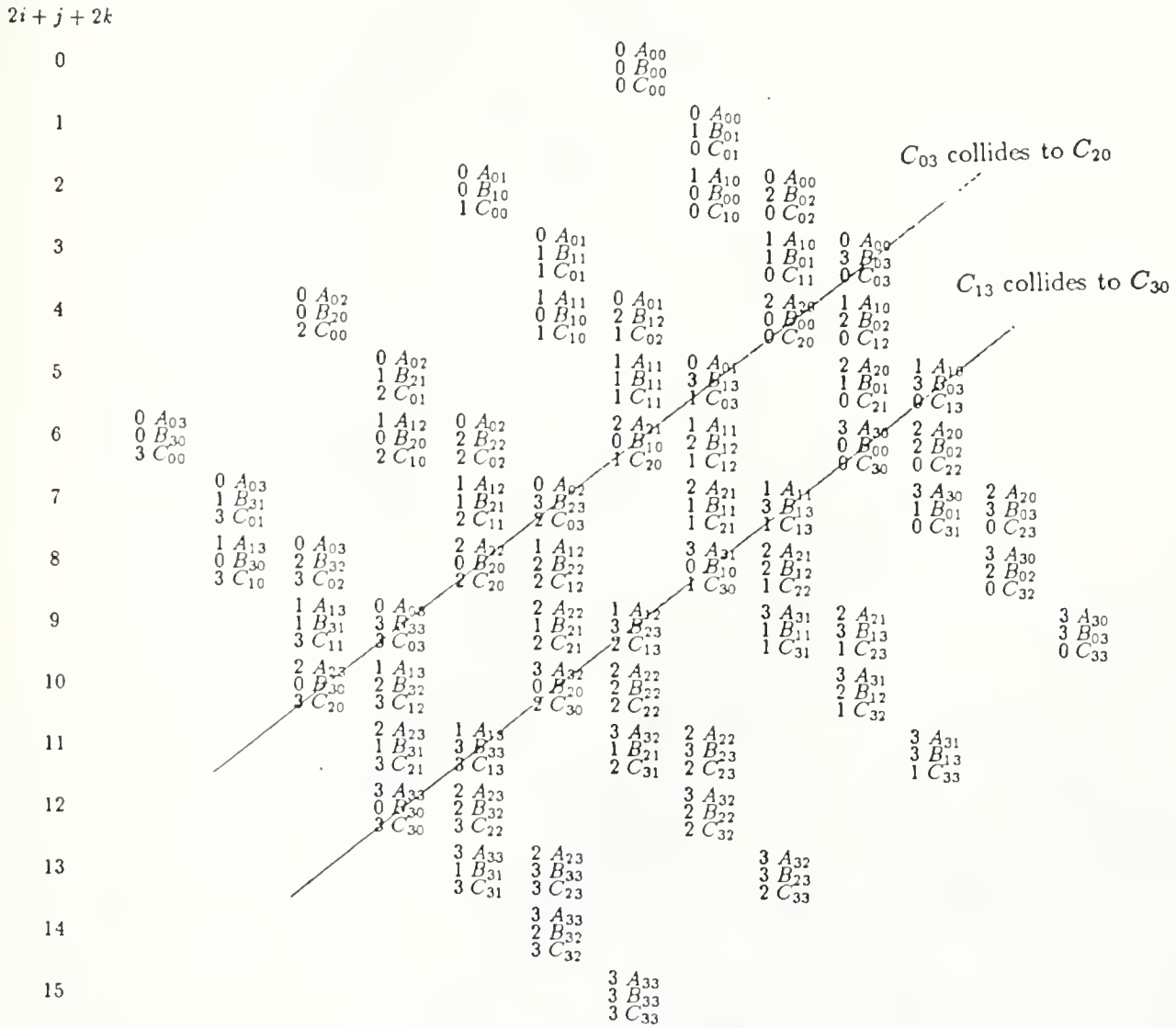
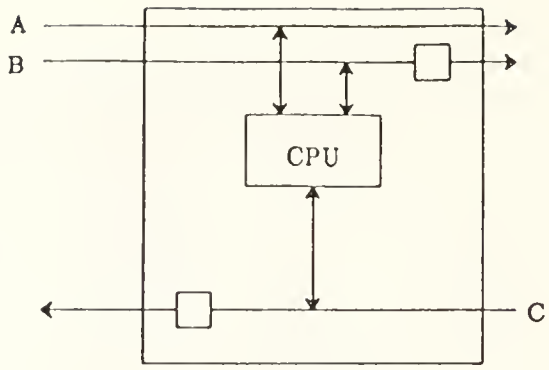


Fig. 7. The data-flow diagram of $\mathbf{H}^1 = (2, 1, 2)$ and $\mathbf{S} = (1, 1, -2)$. Where C_{03} collides to C_{20} and C_{13} collides to C_{30}.



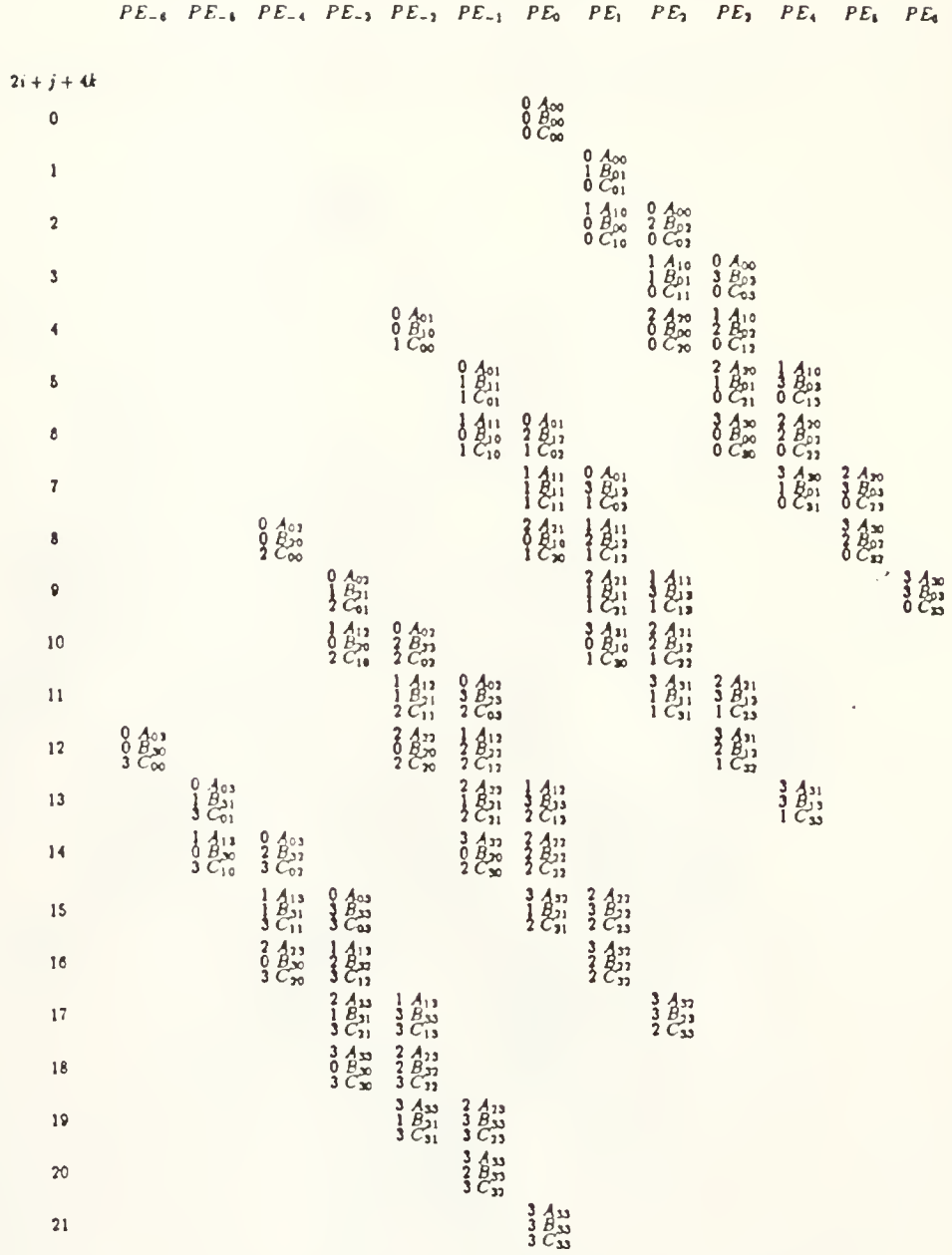
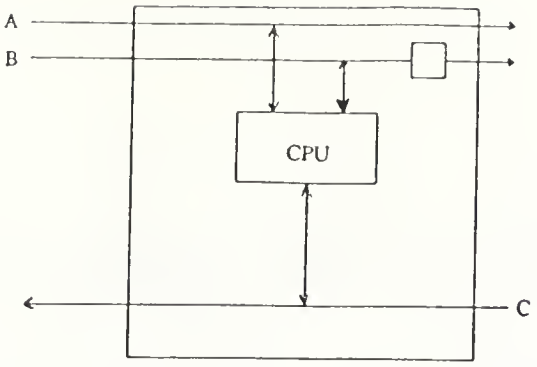


Fig. 8. The data-flow diagram of $H_2^1 = (2, 1, 4)$ and $S_2 = (1, 1, -2)$.



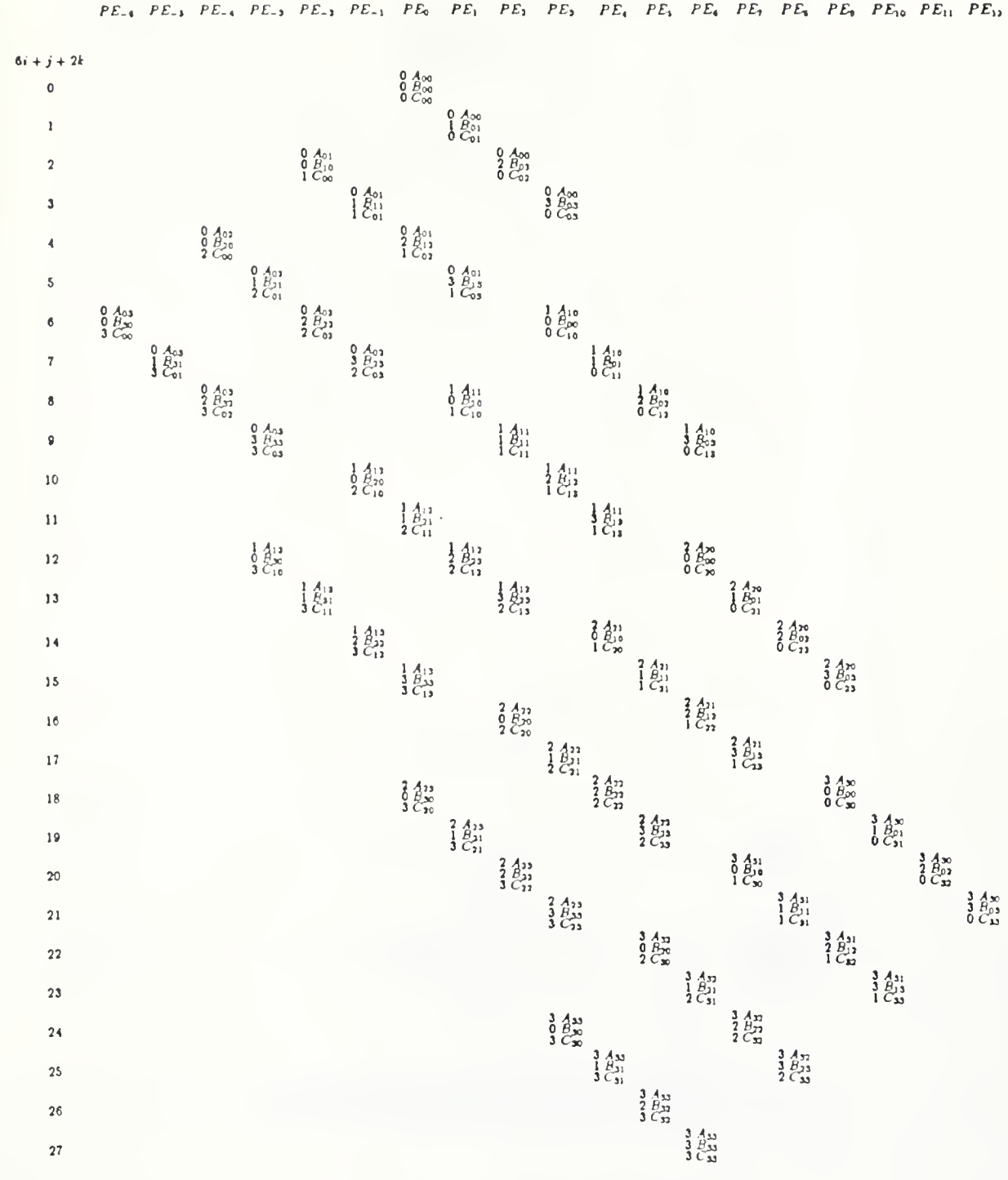


Fig. 9 The data flow diagram of $\Pi_3^1 = (6, 1, 2)$ and $S = (3, 1, -2)$

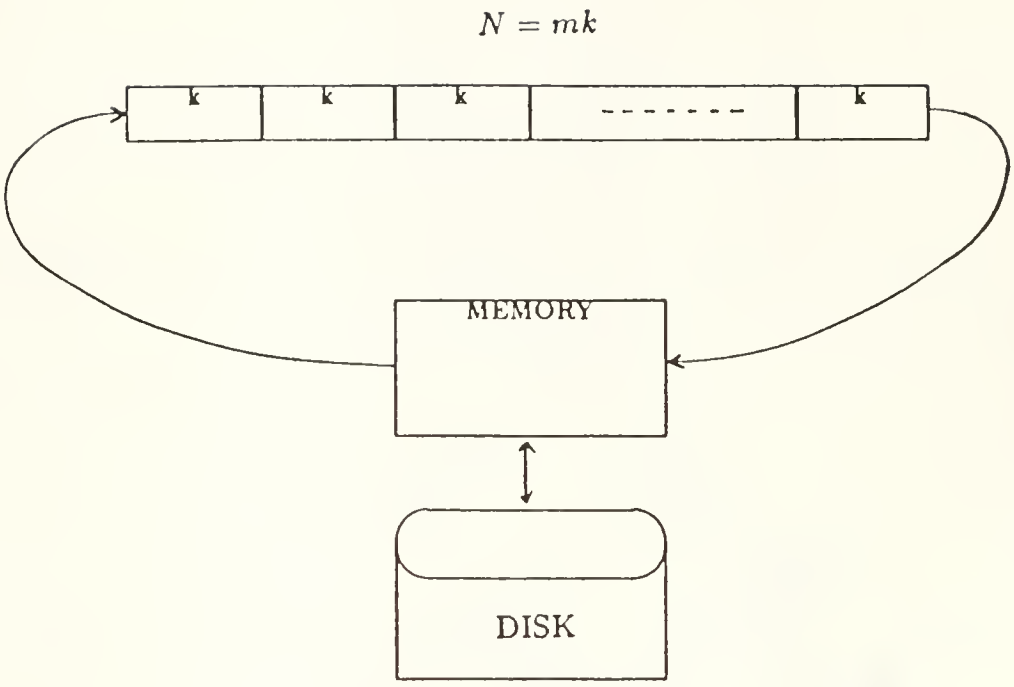


Fig. 10-a. $N = mk$ and data streams are fed into the N-storage array only once.

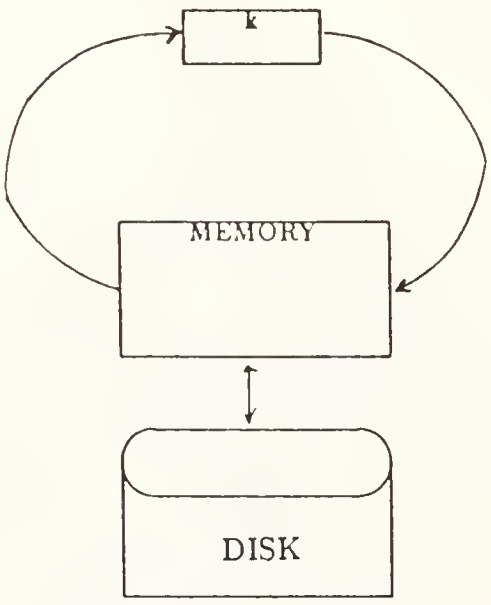


Fig. 10-b. data streams are fed into the k-storage array m times.

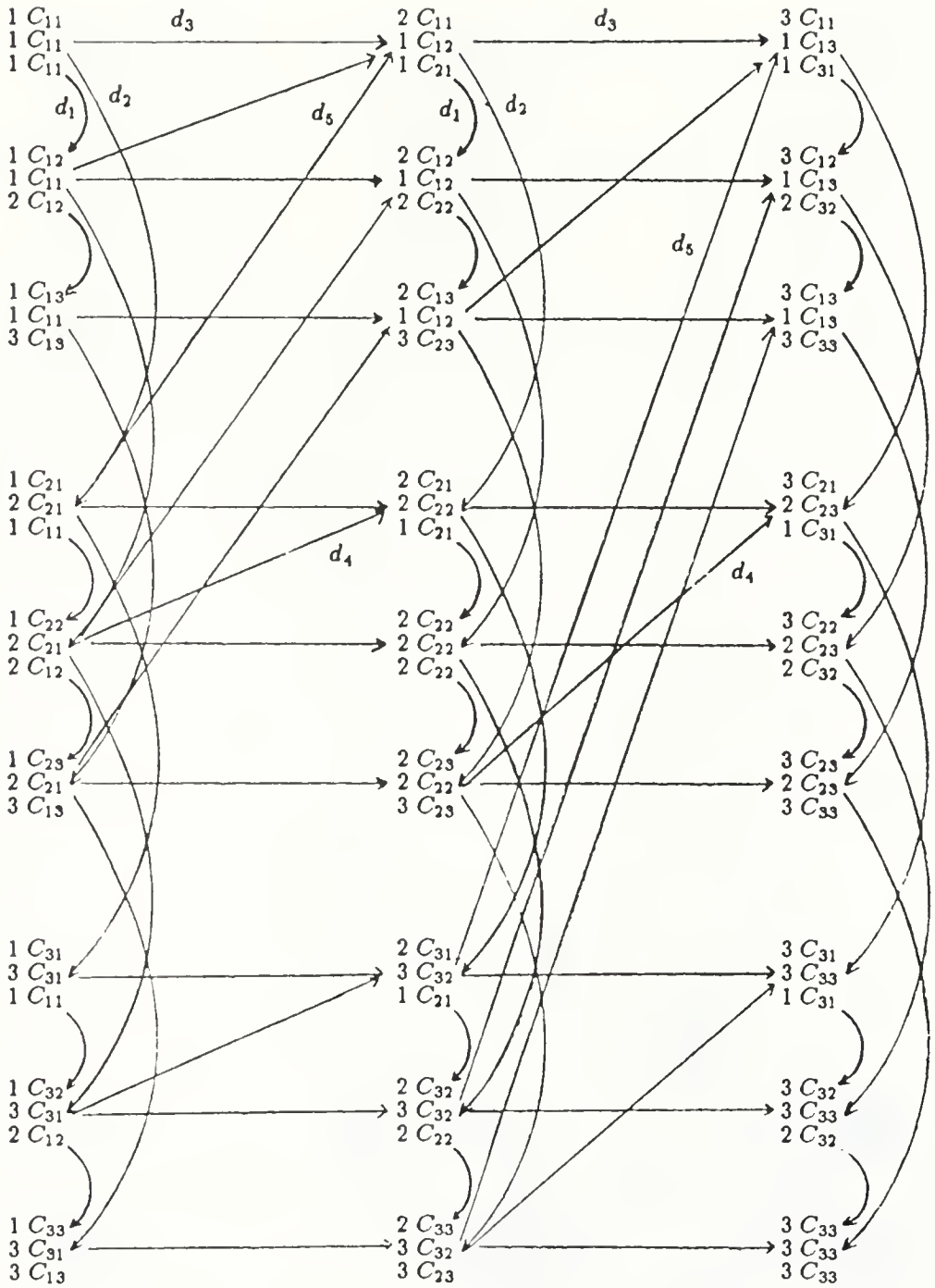


Fig. 11. The data-dependence graph for the Warshall-Floyd algorithm.

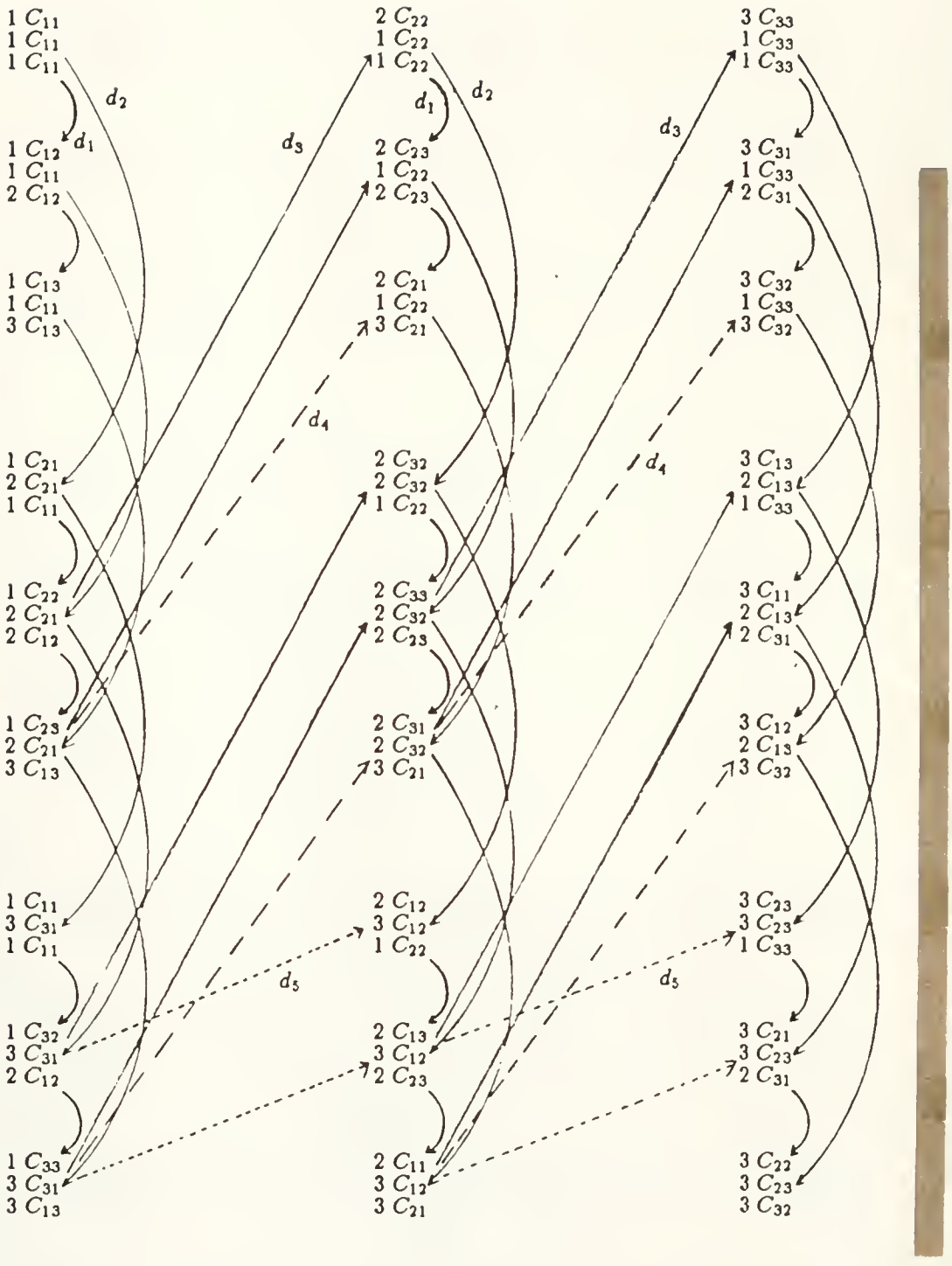
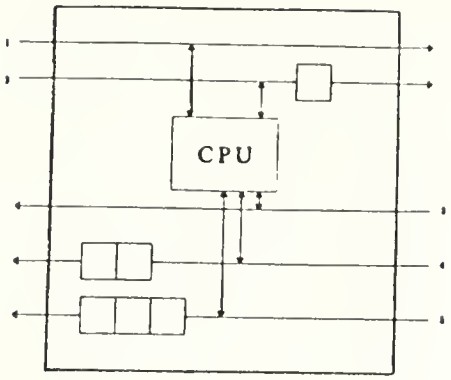


Fig. 12. The data-dependence graph for the reindexed Warshall-Floyd algorithm.



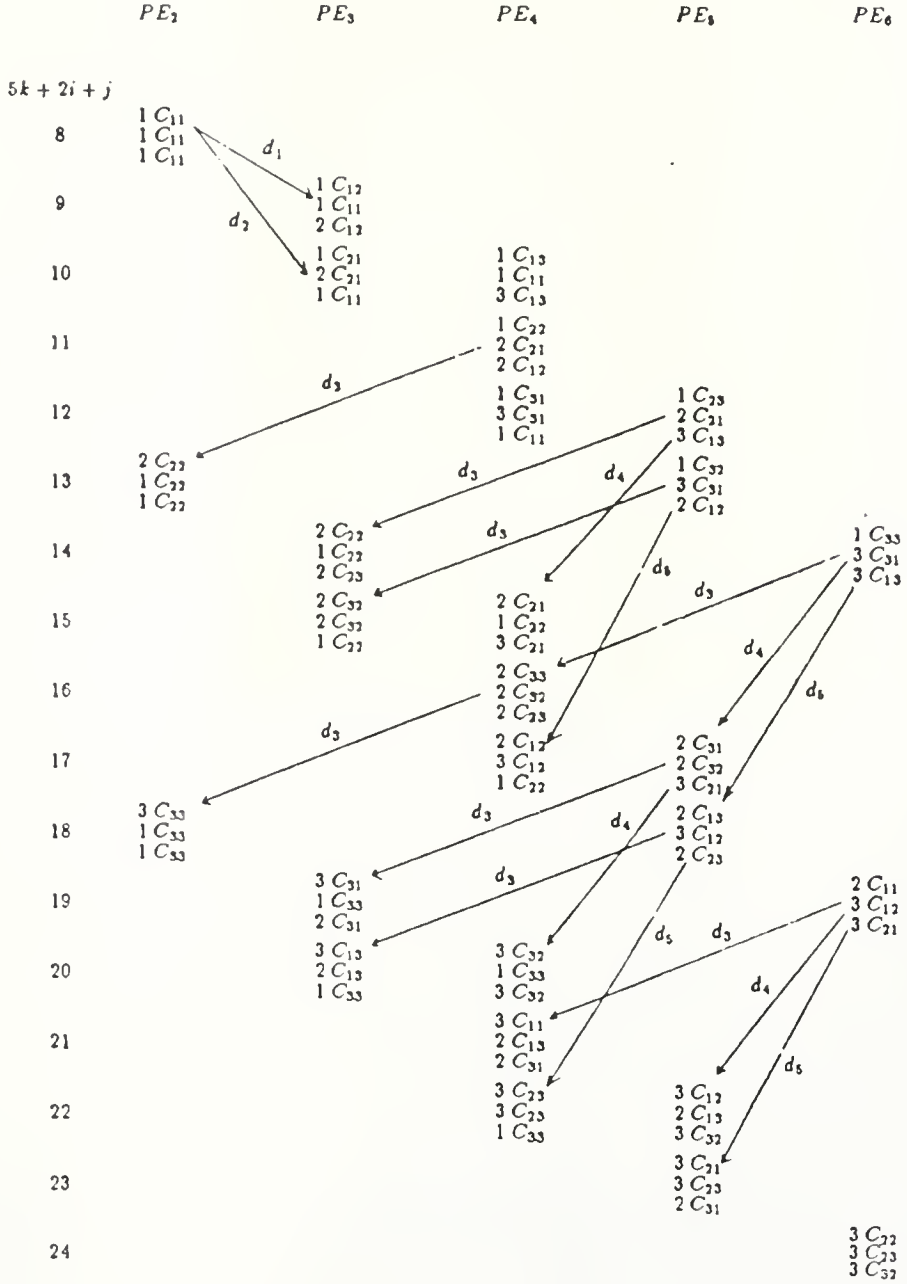


Fig. 13. The data-flow diagram for $\Pi_6^1 = (5, 2, 1)$ and $S_6 = (0, 1, 1)$.

NYU COMPSCI TR-355 c.1
Lee, PeiZhong
Synthesizing linear-array
algorithms from nested for
loop algorithms.